Language Specification for SUGAR

Health Informatics 3

June 18, 2015

1 Introduction

This document will describe the language used to describe analyses in our program. The language should be simple and expressive. This will ensure the best usability and improve the user experience.

We've opted to name her SUGAR, which stands for Simple Usable Great Analysis Reasoner.

2 Basic Idea

We like the BASH approach to piping input from one process to another, so we'll be borrowing the pipe | command. Don't worry, we'll give it back. Parentheses are great to use for grouping things like parameters and the ; is great for closing things.

3 Example

Let's start with something like:

$$def\ idCheck : Constraint = statSensor.id = 170;$$
$$from(statSensor)|constraints(idCheck)|is(person1)$$

Here we run into a couple of things, first of all we have our processes (from, constraints etc.). These are straight forward and should pose no issues. They always have their parameters between parentheses (if they have no parameters parentheses are still required).

Next we have our pipes, which we have described in the previous section. They basically let the output of the one process be the input of the next.

We also encounter a macro, this will be discussed in the next section.

4 Macros

Macros are a way of splitting the processes you want to perform on the data (your process chain) and exactly what that process should do.

Let's begin with another example:

$def\ idCheck : Constraint = measurement.id = 1\ AND\ (measurement.level > 5)\ OR\ NOT(true)\ AND\ (measurement.level > measurement.value)$;

All macros start with the word def. It is followed by the identifier for the macro (which can then be used in a process chain). In this case the identifier is idCheck. After this we see the type. We currently have several types in our language: Constraint, GroupByColumn, GroupByConstraint, Join, Computation, Comparison and Connection.

After the type, we require an = to specify the actual content of the macro. The actual working depends on the type of macro. We close the macro with an ;.

5 Literals

Since all languages require some hardcoding to make everyone's live easier, we also support it. You have several types of literals.

5.1 String literals

Example: "$Hello$"
String literals are denoted by ""s. Unfortunately that means we don't support the use of " in string literals. We encourage any future developers to work on supporting this.

5.2 Boolean literals

Examples: $true$, $false$
Boolean literals are true or false.

5.3 Integer literals

Example: 5
Integer literals are quite simple. They are any number not containing a dot.

5.4 Float literals

Example: 5.0
Float literals are also doable. They are any number containing a dot.

5.5 Date literals

Examples: $\#1995-01-17\#$, $\#13:33\#$, $\#1995-01-23\ 23:35:54\#$

Dates are more tricky. We only support one date format which is year-month-day. This is to prevent ambiguity between the day-month-year and month-day-year formats. The time format is hour:minutes:seconds or hour:minutes. In the latter case the seconds are set to 0.

Dates are always enclosed between hashtags.

Combined the two will result in a datetime.

5.6 Period literals

Examples: $\#5\ DAYS\#$, $\#0\ MOTNHS\#$, $\#5\ YEARS\#$

Periods are a number of days, months or years. This is most useful when adding or subtracting a period of time from a date.

Periods are also enclosed between hashtags and the supported units are DAYS, MONTHS and YEARS.

6 Operators

In our language we support many operations on numbers and booleans.

For numbers we support:

- $+$
- $-$
- $*$
- $/$
- $\hat{}$
- $\%$
- $SQRT$
- $>$
- $<$
- $<=$
- $>=$

For booleans we support:

- AND
- OR
- NOT

All values support the equality operation $=$.

6.1 Date operations

To operate on dates we have several functions which can be called.

6.1.1 Relative

In order to work more freely with dates we support the RELATIVE function. This functions calculates the differences between 2 dates in a given time unit.

For example: $RELATIVE(\#1995-01-17\#, \#2015-06-18\#, DAYS)$ Gives the result 7457.

The first 2 arguments are the dates to compare and the third argument is the unit. The supported units are DAYS, MONTHS and YEARS.

6.1.2 Combine

The combine function combines a date and a time to give a date time.

For example: $COMBINE(\#1995-01-17\#, \#12:12\#)$
Returns the equivalent of $\#1995-01-17\ 12:12\#$.

The first argument is the date and the second the time.

6.1.3 To date and To time

To get either the time part or date part of a given date time you can call TO_DATE or TO_TIME.

For example: $COMBINE(TO\_DATE(\#1995-01-17\ 12:12\#), TO\_TIME(\#1995-01-17\ 12:12\#))$
Results in $\#1995-01-17\ 12:12\#$ and is a rather useless operation.

TO_DATE and TO_TIME both only take one date time argument.

6.1.4 After and Before

To compare dates we have the BEFORE and AFTER operations.

For example: $\#1995-01-17\#\ BEFORE\ \#1995-01-18\#$
Results in true.

All different temporal values are supported, so BEFORE and AFTER work with times, dates and date times. Do note: that mixing only works between dates and date times. Since $\#1995-01-17\#\ BEFORE\ \#12:00\#$ is nonsense.

6.1.5 Add and Min

To add or subtract time we support the ADD and MIN operations.

For example: $\#1995-01-17\#\ ADD\ \#1\ YEARS\#$
Results in $\#1996-01-17\#$.

The left side argument should either be a date or a date time and the right side argument should be a period. The supported units are DAYS, MONTHS and YEARS.

7 Constraints

Constraints can be somewhat complicated and rightfully so, therefore the declaration should be powerfull, but not overly complicated.

In the end a constraint is simply a boolean expression. This means that both the following are valid constraints:

$def\ simple : Constraint = true;$

$def\ complicated : Constraint = NOT(RELATIVE(\#1995-01-15\#, admire2.date, DAYS) > 2);$

8 Computation

To perform a computation on a table and save the result we support the computation analysis. For example:

$def\ comp : Computation = NAME\ relativeDates\ NEW\ SET\ COLUMNS$
$RELATIVE(\#1995-01-17\#, TO\_DATE(admire2.date), DAYS)\ AS\ date;$
$from(admire2)|computation(comp);$

Results in a new table being called relativeDates containing a single column called date in which there is a integer which is the number of days between my birthday and the given measurement.

The computation has the following syntax:

$defIdentifier : Computation = NAME\ Identifier$
$(NEW|INCLUDE\ EXISTING)$
$SET\ COLUMNS\ (Expression\ AS\ Identifier,)+;$

If INCLUDE EXISTING is used instead of NEW, the columns of the original table are included in the result. As shown you can include multiple columns by separating them with a comma.

9 Group by

We support chunking in the form of a group by. We have 2 kinds of group by operations. One is the group by column. You specify a column in which a single value forms a single group. The other is the group by constraint in which you specify a set of constraints, in which each constraint forms a group. For example:

$def\ group : GroupByColumn = NAME\ perday$
$ON\ admire2.date$
$FROM\ MAX(admire2.measurement)\ AS\ max,$
$AVERAGE(admire2.measurement)\ AS\ avg;$
$from(admire2)|groupBy(group)$

$def\ group : GroupByConstraint = NAME\ dayparts$
$ON\ admire2.time\ BEFORE\ \#12:00\#\ AS\ morning,$
$admire2.time\ AFTER\ \#12:00\#\ AS\ afternoon$
$FROM\ MAX(admire2.measurement)\ AS\ max,$
$AVERAGE(admire2.measurement)\ AS\ avg;$
$from(admire2)|groupBy(group)$

The group by column has the following syntax:

$def\ Identifier : GroupByColumn = NAME\ Identifier$
$ON\ ColumnIdentifier$
$FROM\ (Function(ColumnIdentifier)\ AS\ Identifier,)+;$

The group by constraint has the following syntax:

$def\ Identifier : GroupByColumn = NAME\ Identifier$
$ON\ (BooleanExpression\ AS\ Identifier,) +$
$FROM\ (Function\ AS\ Identifier,)+;$

10 Connections

Connections should be directed and should be capable of being many to many. This means implementing such operators:

(First,) Results, Triggered by, Constraints operands.

The first means that we only connect one row to another. The results and triggered by are to be used when checking if there are any connections. The

This results in something like this:

$def\ measurementInput(measurement, input) : Connection = FIRST\ input\ WHERE\ input.time < measurement.time\ AND\ input.date >= measurement.date\ ORDER\ input.date, input.time\ ASC$

$connection(measurementInput, statSensor, website)|is(connected)$

11 Coding

Codings are quite complicated. We want to cover all possible combinations of events. Let's say something like this: $def\ input(input, measurement)\ :\ Coding = IF\ input\ WHERE\ measurement.time < (input.time - 5\ min)\ AND\ measurement.date = input.date\ STORE\ "MI", input.time, input.date$

$coding(input, website, statSensor)|append(coded)$

To code the taking of a measurement and immediately entering it in the website.

So you specify IF there is a certain row, for which there is another row or multiple other rows and then you output a code and certain other values. There should be a way to connect rows and the code back together afterwards.

12 Comparison

TBD

13 Conversion

TBD