VALIDATION AND TRANSFORMATION LANGUAGE – VTL 1.1 Cheat Sheet *(page 1)*

Syntax

Assignment with := symbols – e.g. *var_1 := 12* assigns the number *12* to the variable *var_1*
Assignments within { } (curly brackets) are local, i.e. the variable can be returned only within those brackets
Comments start with */** and end with ***/** – e.g. */* some comments */* can be inserted along the code

Conditional expression:
***if** condition_1 **then** statement_1 **elseif** condition_2 **then** statement_2 **else** condition_3 **then** statement_3*
E.g. the expression: *B := **if** A > 5 **then** 1 **else** 2*
assigns the number *1* to the variable *B* if *A* greater than *5*, otherwise it assigns the number *2* to *B*

Basic variable types

integer (whole number, e.g. *36*)
float (floating-point number, e.g. *36.45*)
boolean (*true* or *false* value)
date (timestamp, e.g. 2016/11/18)
string (any UNICODE string, e.g. "hi!")

Indexing of string characters:
e.g. string_var := "hello" → h e l l o / **0 1 2 3 4**

String operations

Return length: ***length**("hi") ⇨ 2*
To uppercase: ***upper**("hi") ⇨ "HI"*
To lowercase: ***lower**(Hi") ⇨ "hi"*
Trim whitespace: ***trim**(" hi ") ⇨ "hi"*
Join: *"ki"* **||** *"wi" ⇨ "kiwi"*
Slice: ***substr**(str, start, length)*
E.g. if we assign *A := "Hello"*
***substr**(A, 2, 3) ⇨ "llo"*
***substr**(A, 0, 1) ⇨ "H"*
Find: ***instr**(strToSearch, strToFind)*
E.g. if we assign *B := "lo"*
***instr**(A, B) ⇨ 3* (start índex)
Replace: ***replace**(str, old, new)*
E.g. if we assign C := *"ium"*
***replace**(A, B, C) ⇨ "Helium"*
***replace**(A, "ell", "2") ⇨ "H2o"*
Date type from string type:
E.g. if we assign D := *"2016-02"*
***date_from_string**(D, **YYYY-MM**)*
⇨ *2016/02/01* (date type)

Basic dataset structure

Components: (1) Identifiers, (2) Measures, (3) Attributes

Example:

Employee_ID	*Salary*	*Currency*
A1	1000	dollar
B1	1200	euro
C2	800	yen
D2	900	pound

(In real datasets, there can be any number of each component.)

The component titles with corresponding *columns* and *rows* in this table and elsewhere in the descriptions are representational – real datasets may store components and respective data in a different structure.

Input data types for functions and operations

- Scalar; e.g. a simple numeric expression written within the code like *14* or *6.23*
  E.g., *round(6.23, 0)* rounds the number *6.23* to zero decimals (i.e. it returns *6.00*)
- Referenced Measure or Attribute within a dataset; e.g. *ds_bep.obs_value* expression takes the *obs_value* Measure (containing e.g. a list of numbers) from the *ds_bep* dataset
  E.g., *round(ds_bep.obs_value, 0)* rounds all values found in the *obs_value* Measure
- Referenced dataset without selected Measure or Attribute; e.g. *ds_bep*, which takes the entire *ds_bep* dataset and applies the given operations on all values whose data type allows it
  E.g., *round(ds_bep, 0)* rounds all numeric values found in any Measure of the *ds_bep* dataset

As in the examples above, functions and operators can be used, in general, not only on simple scalars, but also on datasets or on given measures within a dataset, in a logical manner. E.g. *dset_1 + dset_2* pairs the rows of the two datasets based on matching Identifier Components, and returns a dataset containing the addition of the Measures of these rows – see *dataset operations* on page 2.

Rulesets

- Datapoint ruleset: rules that apply to individual "rows" in the data, e.g.:
  define datapoint ruleset** wage_curr (Currency, Salary) **is
  *curr_rule: Currency = "euro" **errorcode** "not euro" **errorlevel** 2;*
  *amount_rule_1: **when** Currency = "euro" **then** Salary **between** 1100 **and** 2500;*
  *amount_rule_2: **when** Currency = "dollar" **then** Salary > 1200*
  end datapoint ruleset
  ⇨ *wage_curr* can now be used for validation checks or filtering (see *check()* and *filter()*)
  (When used for validation with the check() function, violation of the *curr_rule* rule returns the given row including the *"not euro"* text under a new "ERRORCODE" component, and the number *2* under an "ERRORLEVEL" component.)

- Hierarchical ruleset: rules that interrelate the contents of "rows" across the data, e.g.:
  define hierarchical ruleset** wage_compare (**variable** = Employee_ID) **is
  *compare_employees: A1 > B1 **errorcode** "A1 lower nominal wage than B1";*
  *compare_sum: C2 = A1 + D2 **errorcode** "sum does not equal C2"*
  end hierarchical ruleset
  ⇨ *wage_compare* can now be used for checks or calculations (*check(), aggregate()*)
  (Again, when used for validation, the appropriate errorcodes are returned.)

Validation check

The *check()* function performs validation checks using (one or more) predefined rulesets.
E.g. ***check**(ds3, wage_curr)* or ***check**(ds_3, wage_compare)* performs the *wage_curr* or *wage_compare* checks, respectively, as described in the given rules, on the *ds3* dataset. By default, only rows that violate the rule are returned, with (optional) errorcode/errorlevel feedback columns.

The *check()* function can also be used with a single in-line rule, e.g. ***check**(ds3.obs_val < 10)* checks each (number) value within the *obs_val* measure of the *ds4* dataset to evaluate whether it is less than 10 – and returns rows in which the value is greater than or equal to the number 10.

The *check_value_domain_subset()* function checks whether the specified components in a dataset respect the restrictions (format, content) given in a *value domain predefinition* (see page 2), in the format of ***check_value_domain_subset**(dataset, components_to_check, domain_val_predefinition).*

Aggregate

The *aggregate()* function aggregates datasets based on applicable rules (equations) in a *hierarchical ruleset*. (All non-applicable details, e.g. errorcodes or boolean inequality, are ignored.)
E.g. ***aggregate**(ds3, wage_compare)* returns the *A1+D2* sum under component *C2* (see *Rulesets*).

For more general aggregate functions, without rulesets, see *avg()*, *max()*, etc. on page 2.

Basic functions and operators

Function	Description	Example(s)
get	Retrieves dataset from a structure. Optional clauses: *keep()*, *dedup()*, *filter()*, *aggregate()*.	***get****("DIR_1/DATAFILE2",* ***keep****(ID1, M2)* retrieves the dataset contained in the "DATASET2" file and returns it with only the ID1 Identifier and M2 Measure components
put	Stores dataset to persistent structure.	***put****(ds1, "DIR2/DFILE.STO")* stores the *ds1* dataset to a file named "DFILE.STO"
[ ] (i.e. join)	Joins datasets based on Identifiers. Optional clauses: *drop()*, *keep()*, *filter()*, etc. See *dataset operations* .	***[****ds1, ds2****]*** *{* ***filter*** *ds2.M4 <> 0 }* returns a dataset that contains all rows with common Identifiers in the ds1 and ds2 datasets, but excluding those with 0 value in ds2
Sets: *union* *intersect* *symdiff* *setdiff*	Set functions merge datasets based on Identifier components: *union()* keeps one of each unique row; *intersect()* keeps only the rows that are common in the input datasets; *symdiff()* keeps all rows that are not common, *setdiff()* keeps one of each row that is not common.	***union****(ds1, ds2)* returns all unique rows once; namely, it returns all rows in the dataset *ds1* and complement it with all rows in the dataset *ds2* that do not have the exact same Identifiers as any of the rows in *ds1* ***symdiff****(ds1, ds2)* returns all the rows in *ds1* that do not have the exact same Identifiers as any of the rows in *ds2*, and also all the rows in *ds2* that do not have the same Identifiers as any of the rows in *ds1*
Aggregate, e.g.: *avg* *count* *max* *min* *sum*	The specified aggregate functions calculate averages, sums, variances, maximum values, ranks, etc., within the specified Measure component. One may "*group by*" or "*along*" one or more Identifier component. The "*time_aggregrate*" aggregates values along a time dimension.	***avg****(ds1.m3)* ***group by*** *time_year* returns the averages of the *m3* measure in the *ds1* dataset grouped by the *time_year* Identifier values (e.g. average of all data from 2009, average of all data from 2011, average of 2012, etc.) ***max****(ds1.m5)* ***along*** *time* disregards the *time* values and return max. values grouped by all other Identifier values ***sum****(ds4)* ***time_aggregate****("Q","A")* transforms all quarterly Measure values in *ds4* into corresponding yearly sums
Analytic, e.g.: *first_value* *last_value* *lag* *rank* *ntile*	Analytic functions use customizable sliding windows that move across the rows of a dataset to calculate the rows of the output dataset, e.g. by moving each original row forwards or backwards, or aggregating the values in several subsequent rows, etc.	***first_value****(ds3)* ***over*** *(****partition by*** *area* ***ordered by*** *time)* for each Measure component in *ds3*, for each value within the same *area* (ID), assigns the 1st value (in 1st row) of the given Measure within that *area*, with rows ordered by *time* ***lag****(ds3,1,5)* ***over*** *(****partition by*** *geo* ***ordered by*** *age)* moves each Measure value within the same *geo* (ID) into the previous row, and replaces every offset (every last row within *geo*) with the number *5,* with rows ordered by *age*

Dataset operations

E.g. *ds_A* is:

Employee	*Salary*	*Currency*	*Benefits*
A1	1000	dollar	200
B1	1200	euro	150
C2	800	yen	270

and *ds_B* is:

Employee	*Salary*	*Currency*	*Benefits*
A1	700	dollar	40
B1	950	euro	0
D2	1100	pound	190

where, in both cases, *Employee* is an Identifier component, *Salary* and *Benefits* are both Measure components, and *Currency* is an Attribute component. Consequently, the exemplary calculations below may be executed. In each of these examples, the components in the resulting dataset keep their names and types, so e.g. the two Measures always both stay Measures (though the values may be changed).

ds_A / 2 (division of the dataset by a simple scalar) returns:

Employee	*Salary*	*Currency*	*Benefits*
A1	500	dollar	100
B1	600	euro	75
C2	400	yen	135

[inner *ds_A, ds_B****]*** *{ ds_A - ds_B }* (or simply: *ds_A - ds_B*) returns:

Employee	*Salary*	*Currency*	*Benefits*
A1	300	dollar	160
B1	250	euro	150

[outer *ds_A, ds_B****]*** *{ ds_A + ds_B }* (or: ***[outer]*** *ds_A + ds_B*) returns:

Employee	*Salary*	*Currency*	*Benefits*
A1	1700	dollar	240
B1	2150	euro	150
C2	*null*	yen	*null*
D2	*null*	pound	*null*

[inner] *{ ds_A + ds_B ,* ***drop*** *Benefits,* ***filter*** *ds_A.Salary < 2000}*) returns:

Employee	*Salary*	*Currency*
B1	2150	euro

Define function

User defined function definition e.g.:
create function *multiply_func(x, y)*
as *x*y*
⇨ *multiply_func(2, 3)* returns 6

Recoding identifier values

The *transcode()* function recodes the values of a given identifier component. E.g. ***transcode****(ds1, ds_map, GEO)* recodes all values in the *GEO* Identifier component of the *ds1* dataset, based on the *ds_map* variable – which can be, in the simplest case, a dataset containing a *MAPS_FROM* Identifier component, and *MAPS_TO* Measure component; the former containing values to be changed, the latter the values to insert (e.g. FRANCE to FR, GERMANY to DE, etc.). The *ds_map* variable can also be a predefined "mapping object"; see *define mapping ruleset* in the Reference Manual.