jVTLlib

Reference Manual

Pier Riccardo Monzo

Versione documento 0.8

Sommario

Cronologia revisioni

Data	*Versione*	*Descrizione*	*Autore* (sigla)
30/03/18	0.1	Versione iniziale	P.R.M.
01/04/18	0.2	Aggiunte clausole	P.R.M.
03/04/18	0.3	Aggiunte prime Funzioni Set	P.R.M
04/04/18	0.4	Aggiunte altre Funzioni gruppo Set e le istruzioni condizionali.	P.R.M.
16/04/18	0.5	Aggiunta clausola drop e funzioni join. Modificata istruzione check. Aggiunto parametro keep a Get. Aggiunti esempi vari.	P.R.M.
02/05/18	0.6	Aggiunti capitoli su Funzioni/Procedure. Riscritta parte iniziale. Modificate le quattro operazioni.	P.R.M.
05/05/18	0.7	Aggiunte funzioni d'aggregazione. Aggiunta funzione replace di stringhe.	P.R.M.
15/05/18	0.8	Correzioni e pulizia finale documento. *Questa versione si può considerare la versione finale per il tirocinio.*	P.R.M.

Struttura del documento

In questo manuale si descrivono in dettagli gli operatori e le funzioni disponibili ed è diviso in due parti.

Nella prima parte, è presente una lista di operatori e funzioni, una lista di tipi di dato implementati, una spiegazione di come questi tipi di dato interagiscono tra di loro e, molto importante per il resto del documento, un capitolo dedicato alle convenzioni usate nella descrizione dei singoli operatori/funzioni.

La seconda parte del documento raggruppa gli operatori/funzioni in capitoli: ognuno rappresenta un gruppo di istruzioni che lavorano nello stesso contesto. Ad esempio, gli operatori matematici sono differenziati da quelli che operano sulle stringhe.

In ognuno di questi capitoli, ogni istruzione sarà descritta nel seguente modo:

- **Descrizione**: una breve descrizione.
- **Sintassi**: il *meta-linguaggio* usato per la descrizione della sintassi è evidenziato nel capitolo apposito.
- **Parametri**: gli eventuali parametri usati dall'istruzione e una descrizione del loro significato.
- **Restrizioni**: i limiti di funzionamento di un'istruzione e una descrizione del comportamento dell'interprete se ignorati.
- **Ritorna:** i valori di ritorno dell'istruzione, se presenti.
- **Semantica:** una descrizione più approfondita dell'istruzione e di come andrebbe usata.
- **Esempi:** per esplicitare l'uso tramite dati.

Tipi di dati primitivi

A differenza delle specifiche originali VTL, quelle implementante in questo progetto sono più aderenti al modello BIRD, seppur in una forma più generica.

Nel modello BIRD, ogni componente viene descritto tramite un dominio e un sottodominio. Nella libreria è presente un file che contiene il mapping completo tra i domini/sottodomini di BIRD e i tipi di VTL.

Nel caso in cui dominio e sottodominio coincidono per una variabile BIRD, allora questa variabile è di tipo generico (ad esempio, MNTR/MNTR si suppone sia Float VTL, laddove MNTR indica un valore monetario reale).

Nel caso invece il sottodominio sia diverso, viene letto un pattern regex che verrà usato in **Value Domain** per validare il sottodominio di un valore. Il tipo del valore viene invece descritto dal dominio.

Caso speciale è per i dominiti *enumerated*, per cui viene creata dinamicamente una lista di valori che verrà usata in un **Value Domain**. In questo caso, il tipo di valore si assume sia sempre String.

I tipi di dato riconosciuti e usati in questa libreria sono i seguenti

Nome	Formato	Dimensione
Integer	Numeri interi con supporto ai segni (+/-)	Equivalente a *Integer* di Java
Float	Numeri con la virgola con supporto ai segni (+/-)	Equivalente a *Double* di Java: numero a virgola mobile a doppia precisione 64 bit.
String	Stringa di testo racchiusa da doppie virgolette (“”)	Equivalente a *String* di Java: limitato dalla memoria assegnata.
Boolean	Valore booleano espresso dai caratteri “T” per true ed “F” per false. Supportati anche le parole chiave true/false per compatibilità	Conservato in memoria come *Boolean* di Java, ma elaborato per supportare valori non standard.
Date	Supportati diversi formati di date: *yyyy-MM-ddTHH:mm:ssZ* *dd/MM/yyyy* *yyyy-MM-dd HH:mm:ss.fff* *yyyy-MM-dd* *yyyy-MM"* *NB: gli orari non vengono supportati.*	Si usa una libreria personalizzata basata su un oggetto del pacchetto *java.time*. Il range temporale dipende dal formato scelto.

Importante notare che questi dati primitivi sono fondamentali per eseguire correttamente le operazioni, siano esse matematiche (Integer e Float), o relazionali (Boolean), ma è altrettanto importante sottolineare che generalmente tutti i dati sono trattati, internamente alla classe Scalar, come se fossero stringhe.

Nota: *per il resto del documento verrà usata spesso la parola **Scalare** per indicare un valore espresso come tipo di dato primitivo.*

Artefatti

Oltre ai dati primitivi, la libreria supporta una serie di oggetti più complessi, descritti tramite il *definition language* del VTL.

Seppur non tutti sono usati a livello di script VTL, questi oggetti sono parte fondamentale dell'elaborazione e sono qui descritti

Nome	Descrizione
DataStructure	Definisce la struttura di un DataSet, nello specifico contiene il Nome, il Dominio e il tipo di ogni colonna.
DataSet	L'elemento fondante di VTL. Contiene i dati raccolti in DataPoint, come descritto dal DataStructure associato
ValueDomain	Descrive un sottoinsieme di uno dei tipi primitivi. Ad esempio si può definire un dominio che comprende solo i numeri interi positivi.
DataPoint RuleSet	Descrive una serie di regole per validare un dataset. Quando si applica ad un dataset, le righe che non rispettano le regole vengono scartate.

Come già detto, di questi artefatti solo uno viene effettivamente usato negli script VTL, DataPoint RuleSet, seppur il suo uso è strettamente legato alla funzione **check(DataPointRuleset)** più avanti descritta.

Tutti gli altri tipi sono effettivamente usati solo durante il caricamento (e lo scaricamento) dei dati da fonti di dati esterni. Non ci sono istruzioni per, ad esempio, popolare un dataset inizializzato direttamente dal codice. Neppure l'operazione di assegnazione può essere usata, in quanto ogni assegnazione sovrascrive il tipo di dato precedente, in quanto VTL non è un linguaggio fortemente tipizzato.

Si consideri il seguente esempio:

```
1. define ValueDomain year("only the year", date restrict YYYY)
2. define DataStructure dstr(integer ID identifier; string NAME identifier; date AGE measure)
3. define DataSet ds2("ds_dstr", dstr)
4. ds1: = get("db1/table1")
```

Alla fine dell'elaborazione, *ds1* non sarà del tipo definito da noi, ma del tipo con cui la tabella *table1* è descritta nella fonte di dati originale. Non avverrà alcuna trasformazione al rigo 4, né verrà lanciato un errore se l'istruzione *get()* torna un DataSet con struttura e/o formato diverso da quello da noi definito.

Questo perché VTL non presuppone controlli sui tipi da parte del parser, di conseguenza l'utente che farà uso di script VTL dovrà necessariamente verificare con estrema cura le operazioni descritte da questo prima di poterlo eseguire, pena il fallimento dello script oppure l'inconsistenza dei dati.

Struttura di uno script VTL

Così come descritto nel manuale utente, si consiglia di seguire una certa convenzione quando si crea uno script VTL.

La prima parte è dedicata alle definizioni: Rule Set, procedure, funzioni è consigliabile dichiararle tutte all'inizio, nell'ordine che si preferisce.

Queste istruzioni vengono caricate in memoria e lì risiedono fino a conclusione dell'esecuzione. L'operazione di creazione di questi oggetti è molto veloce e gli stessi occupano pochissimo spazio.

A seguire si consiglia il posizionamento delle istruzioni **get()**, dato che qualsiasi istruzione da questo momento in poi ha bisogno di Data Set per essere eseguita.

Si ricorda che i Data Set sono immutabili. Quindi tutti i Data Set usati sono ottenuti tramite trasformazioni a partire da altri Data Set. Si può facilmente assumere che i Data Set usati in uno script sono figli, diretti o indiretti dei Data Set ottenuti da **get**.

A questo punto sono presenti in memoria tutti i dati necessari, e si può definire il corpo dello script vero e proprio.

Tutti questi Data Set sono attualmente in memoria. Per renderli persistenti, si usa l'istruzione **put()** per salvarli nella base di dati esterna.

Dato che questa istruzione dipende da codice esterno all'interprete VTL, è consigliabile cercare di isolarle, ponendole alla fine dello script.

Questa è una convenzione, ovviamente, non è obbligatoria. L'unica cosa fondamentale è ricordarsi che ogni istruzione ha bisogno che tutti gli artefatti/oggetti usati devono essere presenti in memoria.

Esempio

Ecco uno script di esempio completo, che effettua la validazione di un Data Set, ottenendone un altro con le sole righe che hanno fallito il controllo.

```
1. define datapoint ruleset CN0010(DT_STTLMNT, DT_INCPTN) {
2.     RL1: when (DT_STTLMNT <> null) then (DT_STTLMNT >= DT_INCPTN) errorcode("FALLIMENTO")
3. }
4.
5. Instrument: = get("Tavola2", keep(DT_STTLMNT, DT_INCPTN))
6. result:= check(Instrument, CN0010)
7.
8. put("TavolaOut", result)
```

Si fa notare come VTL essenzialmente si limiti ad effettuare l'operazione di validazione, senza dire effettivamente all'utente qual è stato il risultato della stessa. Quest'ultima operazione è lasciata all'utente, che dovrà aprire la tabella risultante (nell'esempio, "TavolaOut") e verificare manualmente (o tramite script automatizzato) il risultato.

In questo esempio, la **check** senza parametri scriverà in *Instrument* un Data Set contenente solo le righe che hanno fallito il controllo. Quindi l'utente dovrà aprire "TavolaOut" e contare quante righe sono in essa presenti. Se zero, tutti i Data Point hanno passato il controllo. Se diverso da zero, può analizzarle per capire cosa ha fatto fallire il controllo.

Di nuovo, questo aspetto di VTL può sembrare una debolezza, ma si consideri che è sempre possibile scrivere un nuovo operatore che automatizzi questo controllo, seppur uscendo dalle specifiche standard.

Lista degli operatori e delle funzioni VTL

La seguente lista elenca le istruzioni implementate del VTL-DL, specificando il nome, il tipo di dato su cui lavora, e una breve descrizione.

Nome	Tipo	Descrizione
*define ValueDomain	Information model	Descrive un dominio di valori
*define DataStructure	Information model	Descrive una struttura per un dataset
*define DataSet	Information model	Descrive un dataset, con descrizione esplicita o implicita della struttura
define DataPoint RuleSet	RuleSet	Definisce un insieme di regole per la validazione dei singoli datapoint.

(*) queste istruzioni non sono implementate come VTL, ma l'interprete è capace di costruire questi oggetti internamente.

La seguente lista elenca le istruzioni implementate del VTL-ML, specificando il nome, i tipi di dati su cui lavora, il tipo di sintassi (*funzionale, infix*) e una breve descrizione.

Nome	Categoria	Sintassi	Descrizione
Parentesi tonde (…)	Generico	Funzionale	Indica la precedenza nella valutazione di espressioni
Assegnamento :=	Generico	Infix	Assegna il risultato di un'espressione ad una variabile
Appartenenza .	Generico	Infix	Indica uno specifico componente di un DataSet
as	Generico	Infix	Definisce un alias per componenti e risultati di espressioni.
get	Generico	Funzionale	Recupera un DataSet
put	Generico	Funzionale	Salva un DataSet
Concatenazione ||	Stringhe	Infix	Concatena due stringhe.
substr	Stringhe	Funzionale	Estrae una sottostringa da una stringa.
replace	Stringhe	Funzionale	Sostituisce una sottostringa con un'altra.
Unario +	Numerico	Infix	Lascia il segno inalterato
Unario -	Numerico	Infix	Cambia il segno
+ - * /	Numerico	Infix	Calcola le quattro operazioni su valori numerici

ceil floor	Numerico	Funzionale	Arrotonda un numero all'intero superiore o inferiore più vicino
round	Numerico	Funzionale	Arrotonda un numero all'intero più vicino
abs	Numerico	Funzionale	Valore assoluto
trunc	Numerico	Funzionale	Tronca i decimali di un numero con la virgola al numero specificato
exp	Numerico	Funzionale	Esponenziale
ln	Numerico	Funzionale	Logaritmo naturale
log	Numerico	Funzionale	Logaritmo in base specificata
power	Numerico	Funzionale	Potenza
sqrt	Numerico	Funzionale	Radice quadrata
nroot	Numerico	Funzionale	Radice n-esima
mod	Numerico	Funzionale	Modulo
= < > <= >= <>	Booleano	Infix	Comparazione di valori
in not in	Booleano	Funzionale	Verifica se un valore appartiene ad un insieme specificato
isnull not isnull	Booleano	Funzionale	Verifica se il valore è nullo
and or	Booleano	Infix	Calcola l'AND e OR logico tra due condizioni
check(DataPoint RuleSet)	Validazione	Funzionale	Applica uno o più Ruleset a un DataSet
union	Insiemi	Funzionale	Unione di più DataSet
intersect	Insiemi	Funzionale	Intersezione di più Dataset
symdiff	Insiemi	Funzionale	Differenza simmetrica di due DataSet
setdiff	Insiemi	Funzionale	Differenza di due DataSet
nvl	Condizionale	Funzionale	Sostituisce i null con un valore specificato.
If/then/else	Condizionale	Funzionale	Effettua una scelta in base alle condizioni indicate
rename	Clausola	postfix	Cambia nome e ruolo di un componente di tipo Measure
keep	Clausola	postfix	Altera la struttura di un DataStructure

filter	Clausola	postfix	Rimuove i DataPoint che non rispettano una condizione
calc	Clausola	postfix	Calcola il valore di un componente di una struttura
drop	Clausola	postfix	Altera la struttura di un DataStructure
Inner join	Join	Funzionale	Effettua la join tra due DataSet e applica delle clausole
Left join	Join	Funzionale	Effettua la left join tra due DataSet e applica delle clausole.
Funzioni Aggregazione	Analitiche	Funzionale	Effettua operazioni di aggregazione su un DataSet

Operatori e funzioni applicati ai dati

Molti degli operatori e delle funzioni possono essere applicati sia a scalari che a DataSet, od a una combinazione di questi. Inoltre, singoli componenti di un DataSet possono essere usati per indicare nello specifico su quali colonne operare.

Generalmente, quando si opera su DataSet, almeno un componente *Identifier* deve esistere in entrambe, con lo stesso nome e lo stesso tipo di dati. In questo caso, l'operazione viene effettuata sui componenti *Measure* aventi nome e tipo uguale. Se una di queste condizioni non risulta verificata, allora l'operazione non può avere luogo e si genera un'eccezione, poiché la situazione risulta ambigua. Se invece l'operazione viene effettuata tra un DataSet e uno scalare, l'operazione viene effettuata su tutti i componenti *Measure* dello stesso tipo dello scalare. Nel caso in cui non ci sia una misura dello stesso tipo, viene ritornato il DataSet, senza nessuna modifica.

Per chiarire meglio i comportamenti nei vari casi, qui di seguito è presente una tabella esplicativa.

Si assume che *f* sia una funzione/operatore VTL, *dsX* un dataset (X è un numero per distinguerli), *c* è uno scalare (*c* come *Costante*), *m* è un componente di un dataset (*ds.m* è il componente *m* del dataset *ds*).

Caso	Risultato	Regola	Esempi
f(*c*,*c*)	Scalare	*f* è applicato agli scalari direttamente	1+2 "Hello,"||" world!"
f(*ds*,*c*) *f*(*c*,*ds*)	un DataSet che mantiene tutti i componenti *Identifier* e *Attribute* di *ds*, ma con i soli componenti *Measure* ritornati dall'operatore. Quelli con tipo diverso da *c* vengono scartati.	*f* è applicato a tutte le misure con lo stesso tipo dello scalare. *f* è applicato a tutti i DataPoint di *ds.* La cardinalità del risultate DataSet è la stessa di *ds.*	*ds* + 1 log(*ds*,2)
f(ds1,ds2)	Un DataSet che mantiene tutti i componenti *Identifier* e le misure in comune con i tipo accettato dall'operatore. Tutti i componenti *Measure* con tipo diverso vengono scartati,	*f* è applicato alle misure numeriche in comune. *f* è applicato a tutti i datapoint di *ds1* e *ds2* che hanno i componenti *Identifier* in comune e i tipi di dato dei campi *Measure*.	*ds1* + *ds2* mod(*ds1*,*ds2*)

	insieme ai componenti *Attribute* di entrambi i DataSet.	La cardinalità del risultate DataSet è data dal numero di DataPoint che rispettano le precedenti regole.	
f(ds.m,c) *f(c,ds.m)*	Un DataSet con tutti i componenti *Identifier* e *Attribute*, ma con il solo campo *Measure* specificato.	*f* è applicato al componente *Measure* specificato *f* è applicato a tutti i DataPoint di *ds*. La cardinalità del risultate DataSet è la stessa di *ds*.	trunc(*ds.value,*1) *ds.ext*||".txt"
f(ds1.m1, ds2.m1)	Un DataSet che mantiene tutti i componenti *Identifier* (senza duplicati) di *ds1* e *ds2* e il componente *Measure m1*. La stessa misura *m1* deve essere selezionata in entrambi i DataSet. Gli attributi verranno ignorati totalmente,	*f* è applicato alla misura specificata. *f* è applicata a tutti i DataPoint che corrispondono (hanno un componente *Indentifier* in comune con lo stesso valore) La cardinalità del risultate DataSet è data dal numero di DataPoint che rispettano le precedenti regole.	*ds1.m1* + *ds2.m1* mod(*ds1*.m1,*ds2.m1)*

Seppur esista l'operatore *as*, che permette di rinominare *on-the-fly* un componente, e quindi in teoria sia possibile eseguire la valutazione di un operatore su componenti con nomi diversi, ad es.:

```
1. ds1.value + (ds2.val as value)
```

nella pratica questo tipo di istruzione non è contemplato. Una alternativa è quella di usare la clausola *rename* per rinominare uno o più componenti prima di una trasformazione.

Anche se non se ne è fatto cenno nella tabella precedente, uno scalare può essere sia specificato direttamente (e in questo caso prende il nome di *literal*, letterale), sia attraverso una variabile.

Alcune tra le funzioni qui di seguito specificate prendono scalari come parametri per le opzioni. Fare quindi molta attenzione alla sintassi, perché non tutte le funzioni accettano variabili Scalari, ma solo *literal*.

Ordine di valutazione degli operatori

Dato che è possibile mischiare insieme diversi operatori, è importante conoscere l'ordine con cui questi vengono valutati.

Non è stata apportata alcuna modifica alle specifiche originali, quindi di seguito è presente la tabella presa dal *VTL1.1 Reference Manual*.

Order	Operator	Description	Associativity
I	()	Round parenthesis. To alter the default order.	Left-to-right
II	All VTL functional operators	The majority of the operators of the VTL	Left-to-right
III	Clauses and membership	The majority of the operators of the VTL	Left-to-right
IV	unary plus unary minus not	Unary minus Unary plus Logical negation	Right-to-left
V	*	Multiplication	Left-to-right

	/	Division	
VI	+ -	Addition Subtraction	Left-to-right
VII	>, >= <, <= in, not in between	Greater than Less than In (not in) a value list In a range	Left-to-right
VIII	exists_in not_exists_in exists_in_all not_exists_in_all	Identifiers matching	Left-to-right
IX	= <>	Equal-to Not-equal-to	Left-to-right
X	and	Logical AND	Left-to-right
XI	or xor	Logical OR Logical XOR	Left-to-right
XII	if-then-else	Conditional (if-then-else)	Right-to-left
XIII	:=	Assignment	Right-to-left

L'ordine qui riportato potrebbe non essere quello effettivamente considerato dal parser di jVTLlib.

Se l'ordine conta particolarmente, è consigliabile testarlo preventivamente tramite una serie di istruzioni di prova.

Si sottolinea comunque che:

- L'assegnazione := è ovviamente eseguita sempre alla fine, dato che l'espressione deve essere valutata prima di eseguire questo operatore.
- Le parentesi e gli operatori unari (+, -, not) vengono comunque sempre eseguite dopo la valutazione dell'espressione a cui si fa riferimento, che è quella immediatamente a destra dell'operatore.
  NB: [-5+2] è diverso da [-(5+2)], dato che nel primo caso l'espressione è 5, mentre nel secondo è (5+2).
- Le clausole vengono eseguite sempre nell'ordine in cui appaiono, quindi dall'alto verso il basso.
- If/the/else è un operatore che viene usato a sé stante.

Se si è in dubbio, la cosa migliore risulta sempre quello di usare le parentesi tonde per suddividere i blocchi delle espressioni e quindi dare priorità a parti di essa rispetto ad altre.

Convenzioni per la sintassi

Dopo questo capitolo inizia la lunga lista delle descrizioni di ogni singola operazione. Per fare ciò, si farà uso di una sintassi particolare per meglio evidenziare le caratteristiche di ogni singolo operatore.

Questa sintassi è una convenzione specificatamente preparata per questo documento. Non è quindi parte del VTL e non è in un formato standard.

Prima di tutto alcune regole generali:

1. I nomi degli **operatori** e dei **parametri** (*laddove siano sotto forma di parole chiave o flag*) sono **Case-Sensitive**, quindi tengono conto di maiuscole e minuscole.
2. I nomi degli operatori, delle funzioni e di particolari parole chiavi sono evidenziate in **grassetto**. Se ci sono parentesi obbligatorie, come nel caso delle funzioni, anche queste saranno evidenziate in **grassetto**.
3. I parametri, opzionali o non, che indicano variabili, sono indicati in *corsivo*.
4. Si fa uso di schemi simile ad espressioni regolari per indicare la cardinalità di alcuni tipi di parametri.
5. Per indicare il tipo di componenti si fa uso di <IDENT>, <MEAS>, <ATTR> rispettivamente per *Identifier, Measure* e *Attribute.*
6. Per indicare il tipo di dati si fa uso di *Scalar-type* per gli scalari generici e di *X-literal* per indicare un letterale di tipo *X* (*String, Integer, Float, Boolean e Date*).

Per quanto riguarda la cardinalità di oggetti, si usano i seguenti schemi:

- *{…}* : le parantesi graffe servono per raggruppare sotto-espressioni.
- *{opt}?* : indica un oggetto opzionale. In generale si usa il punto interrogativo per indicare che di quell'espressione ne possiamo trovare **zero o una**.
- *{obj}+* : il più indica che quell'oggetto può essere presente **una o più** di una volta.
- *{obj}** : l'asterisco indica che quell'oggetto può essere presente da **0 a più** di una volta.
- *[alt1 | alt2 | alt3]* : le parentesi quadre con il simbolo | (*pipe*) indicano delle alternative. Esattamente **una** delle alternative sarà presente.
- *[alt1 | alt2 | alt3]+* : indica che può essere presente **una o più** alternative tra quelle elencate.
- *[alt1 | alt2 | alt3]** : indica che può essere presente **zero o più** alternative tra quelle elencate.
- Quando sono presenti delle alternative, se si vuole indicare una alternativa come valore di **default**, questa sarà sottolineata. Usata in quei casi in cui un parametro è composto da diverse opzioni, che, nel caso in cui non viene usato, si assume che vale una delle alternative.

Se in una sintassi è presente un blocco ripetuto, questo può essere indicato con un nome generico (*in corsivo*) e poi esplicitato successivamente.

In questo caso si usa il simbolo *::=* (doppi due punti e uguale) per indicare come una variabile è composta.

Questa meta-sintassi è mutuata da BNF (Backus-Naur Form), ma è stata adattata e modificata per la scrittura di questo documento.

Operatori e funzioni

RuleSet VTL-DL

define datapoint ruleset

Descrizione

Definisce un insieme di regole che dovranno essere applicate ai DataPoint di un DataSet.

Sintassi

define datapoint ruleset *rulesetID* **(** *variable {, variable}** **) { ** Rule {, Rule}** **}**

Rule ::= ruleID **: when** [**true** | *antecedentCond*] **then** *consequentCond* {**errorcode(** *errMsg* **)**}?

Parametri

- *rulesetID* è il nome del ruleset.
- *variable* nome della colonna del dataset che verrà presa in considerazione nelle regole.
- *Rule* una singola regola del set, così composta:
  - *ruleID* il nome della regola. Quando usata in una **check()**, verrà creata una colonna **RULE_ID** in cui viene concatenato il nome del RulesSet e l'ID della regola (questo), ma questo solo se la regola è fallita., vuoto altrimenti.
  - *antecedentCond* un'espressione condizionale che deve essere true per passare a valutare la successiva. Se ritorna False, la regola viene ignorata, ritornando comunque **true**. Può essere sostituita dal letterale booleano **true**, in quel caso si passa a valutare la seconda parte della regola.
  - *consequentCond* espressione condizionale. Se questa viene valutata **true**, allora la regola è considerata verificata. Non può essere un letterale booleano.
  - *errMsg* se viene specificato **errorcode**, allora questo può essere un codice numerico o una stringa. In entrambi i casi indica perché quella regola è fallita. Questo parametro verrà inserito in una colonna apposita del dataset.

Restrizioni

antecedentCond e *consequentCond* sono espressioni che devono ritornare obbligatoriamente un booleano.

Ritorna

Un artefatto DataPoint Ruleset, identificato da *rulesetID*.

Semantica

Un RuleSet è utile quando si vuole validare le righe di un DataSet, tramite la funzione *check*. Se per una regola non è necessaria una precondizione, è possibile impostare l'antecedente a true.

Esempi

```
1. define datapoint ruleset rs2(NAME, AGE) {
2.     RL21: when NAME="peppe" then not isnull(AGE) errorcode("regola 21 fallita")
3.     RL22: when true then AGE > 18 and AGE < 22 errorcode("regola 22 fallita")
4. }
```

Operatori e funzioni generici

Parentesi tonde ( )

Descrizione

Le parentesi servono a cambiare la priorità di esecuzione degli operatori in una espressione.

Sintassi

(*espressione* **)**

Restrizioni

Nessuna.

Ritorna

Il valore dell'espressione.

Semantica

Le parentesi si limitano a ritornare il valore calcolato dall'espressione in esse contenute. Si consiglia di non lesinare con il loro uso, per aumentare la leggibilità del codice e per aiutare l'interprete nel caso di ambiguità (che comunque non dovrebbero esistere dato che la grammatica è stata ben formalizzata).

Inoltre, vanno usate per raggruppare parti di espressioni per cambiare il valore delle stesse.

Esempi

```
1. X:= - 5 - 4    // X = -9
2. Y:= -(5 - 4)   // Y = -1
```

Assegnamento :=

Descrizione

Serve ad assegnare ad una variabile il valore di un'espressione

Sintassi

variable **:=** *expression*

Parametri

- *variable* la variabile che ospiterà il valore dell'espressione. Se non esiste, verrà aggiunta in memoria, altrimenti verrà sovrascritto il valore già esistente.
- *expression* l'espressione da valutare.

Restrizioni

Nessuna.

Semantica

L'espressione potrebbe tornare un qualsiasi tipo di dato. L'operazione di assegnazione è distruttiva a sinistra, cioè qualsiasi cosa era contenuta prima nella variabile andrà persa.

Esempi

```
1. a := 3.14*2
2. ds := get("db1/table1")
3. b := substr("stringa", 0, 2)
```

Appartenenza .

Descrizione

Permette di indicare un singolo componente di un DataSet

Sintassi

ds.comp

Parametri

- *ds* DataSet
- *comp* componente del DataSet

Restrizioni

comp deve essere un componente di un DataSet esistente, altrimenti ritorna un'eccezione.

Ritorna

Un oggetto che collega quella colonna a quel DataSet.

Semantica

A differenza delle specifiche, questo operatore non viene mai gestito come operatore generico, ma solo in particolari situazioni. In ogni caso ritorna un oggetto speciale che non fa altro che collegare la colonna con un dataset.

Per sapere quale operatore o funzione supporta questo operatore, leggere le specifiche del rispettivo operatore in questo documento.

Esempi

attualmente usato solo nelle espressioni matematiche

```
1.  ds2 := ds.age + 1
```

get

Descrizione

Recupera un DataSet da una fonte di dati fisica.

Sintassi

get(*dsID {,* **keep(** *column {, column}** **)}?* **)**

Parametri

- *dsID* è una stringa che serve ad individuare una specifica fonte di dati (ad es. una tabella in un database).
- **keep** è un parametro opzionale che indica quali colonne mantenere.
  - *column* è un riferimento valido ad una colonna della fonte di dati.

Restrizioni

dsID deve indicare obbligatoriamente una fonte di dati esistente, altrimenti verrà lanciata un'eccezione.

Ritorna

Un dataset persistente con, eventualmente, le colonne indicate da **keep**.

Semantica

Il funzionamento di **get** è nascosto all'utente. I controlli sui tipi e la creazione del DataSet avviene a livello di codice. Per ulteriori informazioni consultare la documentazione del codice.

Esempi

```
1.  ds1:= get("db1/table1")
2.  ds2:= get("db1/table2", keep(ID, NAME, AGE))
```

put

Descrizione

Salva il DataSet in una fonte di dati fisica.

Sintassi

put(*dsID, ds***)**

Parametri

- *ds* è il dataset che verrà reso persistente
- *dsID* è una stringa che serve ad individuare una specifica fonte di dati.

Restrizioni

dsID deve indicare obbligatoriamente una fonte di dati esistente, altrimenti verrà lanciata un'eccezione.

Semantica

Il funzionamento di **put** è nascosto all'utente. I controlli sui tipi e il salvataggio del DataSet avviene a livello di codice. Per ulteriori informazioni consultare la documentazione del codice.

Esempi

```
1. put("db1/table1", ds1)
```

Operatori e funzioni su Stringhe

Concatenazione ||

Descrizione

Serve per concatenare due stringhe

Sintassi

variable **||** *variable*

Parametri

- *variable* uno scalare o un letterale stringa.

Restrizioni

Non supporta DataSet, ma può essere usato nelle clausole o all'interno di condizioni.

Seppur non esegua controlli sulla presenza dei parametri, viene effettuato un controllo sul tipo dei parametri. Entrambi devono essere stringhe, in caso contrario viene lanciata un'eccezione.

Non esiste infatti la possibilità di fare il cast dei valori.

Ritorna

Uno scalare stringa che contiene la concatenazione delle due variabili in input.

Semantica

Questo operatore prende in input due stringhe che possono anche essere state generate da espressioni più complesse. Per il corretto ordine delle operazioni è consigliato di usare le parentesi tonde.

Esempi

```
1. A:= "Hello, " || "world!" //A = "Hello, world!"
2.
3. X:= "Hello,"
4. Y:= "world!"
5. Z:= X || Y //Z = "Hello, world!"
```

substr

Descrizione

Estrae una specifica sottostringa dalla stringa passata come parametro.

Sintassi

substr(*variable, startPos {, lenght}?* **)**

Parametri

- *variable* uno scalare o un letterale stringa.

- *startPos* un letterale intero, l'indice di partenza della sottostringa.
- *lenght* un letterale intero, la dimensione della sottostringa.

Restrizioni

variable deve essere una stringa, *start* e *end* devono essere dei letterali interi. Non sono attualmente supportati variabili o espressioni per gli indici, che devono quindi essere espressi direttamente.

startPos non può essere minore di zero o maggiore della dimensione della stringa di partenza.

lenght non può essere minore di zero o maggiore della dimensione della stringa di partenza. Inoltre, la somma tra *startPos* e *lenght* non può essere maggiore della dimensione della stringa. In questo caso, il parametro *lenght* viene ignorato.

Ritorna

Una stringa che contiene la sottostringa cercata.

Esempi

```
1. A:= ”Hello, world!”
2. B:= substr(A, 2) //B = "llo, world!"
3. C:= substr(A, 2, 5) //C = "llo, "
4. D:= substr(A, 0, 4) //D = "Hell"
```

replace

Descrizione

Sostituisce una sottostringa con un'altra nella stringa specificata.

Sintassi

replace(*variable, oldchars, newchars* **)**

Parametri

- *variable* uno scalare o un letterale stringa.
- *oldchars* un letterale stringa, indica la sottostringa che deve essere sostituita.
- *newchars* un letterale stringa, la sottostringa che va inserita.

Restrizioni

Tutti i parametri devono essere stringhe.

Il comportamento in caso di stringhe non presenti o cose del genere è regolato dall'istruzione *replaceAll* di Java.

Ritorna

La stringa di partenza se *oldchar* non è presente in *variable*.

La stringa con tutte le istanze di *oldchar* cambiate a *newchar* altrimenti.

Esempi

```
1. A:= ”Hello, world!”
2. B:= replace(A, “ello”, ”i”) //B = "Hi, world!"
```

Operatori e funzioni Numerici

Unario +

Descrizione

Lascia inalterato il segno.

Sintassi

+ *expression*

Parametri

- *expression* un'espressione numerica

Restrizioni

expression deve essere di tipo numerico.

Ritorna

il valore dell'espressione con segno inalterato.

Esempi

```
1.  B:= -5
2.  A:= +B //A=-5
```

Unario -

Descrizione

Inverte il segno.

Sintassi

- *expression*

Parametri

- *expression* un'espressione numerica

Restrizioni

expression deve essere di tipo numerico.

Ritorna

il valore dell'espressione con segno invertito.

Esempi

```
3.  B:= 5
4.  A:= -B //A=-5
```

+, -, *, /

Descrizione

Le quattro operazioni.

Sintassi

ds1 [**+** | **-** | ***** | **/**] *ds2*

ds ::= [*ds.m* | *variable* | *integer-literal*]

Parametri

ds1,ds2 nel caso della somma, moltiplicazione e divisione devono essere valori numerici. Con la sola eccezione della sottrazione che può operare anche sulle Date.

Restrizioni

Nel caso della divisione, *ds2* non può valere 0, altrimenti verrà lanciata una **DivisionByZeroException**.

Entrambi i valori devono essere dello stesso tipo.

Ritorna

Se entrambi scalari, uno scalare numerico con il risultato dell'operazione.

Nel caso in cui si opera su date e si usa la sottrazione, verrà ritornato un intero con in numero di giorni che intercorre tra le date.

Se Scalare e Colonna, il DataSet a cui appartiene la colonna, ma con quest'ultima modificata dal risultato dell'operazione. La cardinalità sarà pari a quella del DataSet di partenza.

Se entrambi DataSet, il DataSet con i campi Measure che sono stati modificati dall'operazione.

Semantica

Bisogna fare attenzione al tipo numerico usato nelle operazioni. Nel caso della divisione e di operandi interi, il risultato non terrà ovviamente conto dei decimali, che verranno scartati.

Nei casi di operazioni tra interi e float, il risultato sarà sempre un float. In questo modo non si perde precisione.

Esempi

ds1		
ID	**NAME**	**AGE**
0	Marco	36
1	Matteo	12
2	Michele	30

```
1.  ds2:= ds1.AGE + 1
```

ds2		
ID	**NAME**	**AGE**
0	Marco	37
1	Matteo	13
2	Michele	31

round, ceil, floor

Descrizione

Arrotonda un float all'intero superiore/inferiore più vicino.

Sintassi

[**round(***variable,decimal***)** | **ceil(***variable***)** | **floor(***variable***)**]

Parametri

- *variable* scalare o letterale numerico, è il valore che va arrotondato
- *decimal* letterale numerico, è il numero di cifre decimali da mantenere dopo l'arrotondamento.

Restrizioni

Variable e decimal devono essere ovviamente numerici, ma in particolare *decimal* deve essere un letterale intero non inferiore a zero.

Ritorna

Uno scalare numerico con il risultato dell'arrotondamento.

Semantica

Se **ceil,floor** ritornano un intero, **round** ritorna un float, la cui precisione dipende molto dal metodo di arrotondamento scelto.

Dato che viene usato un Double per l'operazione, si perde di precisione. Una conversione a BigDecimal è prevista. Non fare affidamento a queste funzioni se si cerca precisione.

Esempi

```
a:= 3.14596
b:= -100.675
r1:= ceil(b) //r1=100
r2:= floor(b) //r2=101
r3:= round(a, 1) //r3= 3.1
r4:= round(b, 2) //r4= -100.68
r5:= round(a, 3) //r5= 3.146
r6:= round(b, 3) //r6= -100.675
r7:= round(a, 7) //r7= 3.14596
```

abs

Descrizione

Valore assoluto di un numero

Sintassi

abs(*variable***)**

Parametri

- *variable* Scalare o letterale numerico.

Ritorna

Uno scalare numerico, dello stesso tipo di *variable*.

Esempi

```
b:= -100.675
r:= abs(b) //r=100.675
```

trunc

Descrizione

Tronca i decimali di un float senza arrotondamento.

Sintassi

trunc(*variable,decimals***)**

Parametri

- *variable* scalare o letterale numerico, è il valore che va arrotondato.
- *decimals* letterale intero, il numero di decimali da mantenere.

Restrizioni

decimals deve essere un letterale intero.

Ritorna

Uno scalare numerico, dello stesso tipo di *variable*.

Semantica

Nel caso in cui si indichino un numero di decimali che sia superiore o uguale al numero di decimali presenti, verrà ritornato il valore originale, senza alterazioni.

Esempi

```
1. a:= 3.14596
2. r:= trunc(a,3) //r=3.14
```

exp

Descrizione

Calcola la potenza in base *e*.

Sintassi

exp(*variable***)**

Parametri

- *variable* scalare o letterale numerico, è la potenza.

Restrizioni

variable deve essere un valore numerico.

Ritorna

La potenza in base *e* con esponente *variable*.

Esempi

```
1. c:= 3
2. r:= exp(c) //r=20
```

ln, log

Descrizione

Logaritmo naturale e in base.

Sintassi

ln(*variable***)**

log(*variable,base***)**

Parametri

- *variable* scalare o letterale numerico, è il parametro del logaritmo.
- *base* letterale numerico, è la base per il logaritmo.

Restrizioni

variable deve essere diverso da zero.

Ritorna

Uno scalare numerico dello stesso tipo di *variable.* Questo significa che conviene convertire un intero in float prima di eseguire la funzione, perché altrimenti verrebbe ritornata un arrotondamento.

Semantica

Non esistendo un metodo diretto per calcolare il logaritmo con base diversa da 10 ed *e*, si utilizza un'approssimazione che potrebbe non ritornare il valore corretto.

Si calcola il rapporto dei logaritmi del valore e della base, quindi se chiamiamo la funzione come

```
1. val := 12
2. z := log(val, 2)
```

Il risultato sarà calcolato come

$$\log_2 12 = \frac{\ln 12}{\ln 2}$$

Che darà quindi lo stesso risultato.

Per derivare questa eguaglianza, si è fatto uso di wolfram alpha, potente motore computazione basato sull'intelligenza artificiale.

Esempi

```
1. e:= 3.0
2. d:= 3
3. r:= ln(e) //r= 1.0986123085021973
4. r:= ln(d) //r= 1
```

nb: manca l'esempio con log(a,b)

power

Descrizione

Calcola la potenza di un numero.

Sintassi

power(*base,exp***)**

Parametri

- *base* scalare o letterale numerico, indica la base della potenza.
- *exp* letterale numerico, rappresenta la potenza.

Restrizioni

Entrambi i parametri devono essere valori numerici.

Ritorna

Il valore di ritorno avrà lo stesso tipo della base.

Semantica

Dato che al tipo del valore di ritorno verrà assegnato il tipo della base, si deve verificare che il campo di destinazione sia dello stesso tipo.

Esempi

```
1. c:= 3
2. r:= power(c,3) //r=27
```

sqrt, nroot

Descrizione

Calcola la radice quadrata o n-esima di un numero.

Sintassi

sqrt(*variable***)**

nroot(*variable,radix***)**

Parametri

- *variable* scalare o letterale numerico, indica il contenuto della radice.
- *radix* letterale numerico, indica l'esponente della radice.

Restrizioni

Entrambi i parametri devono essere valori numerici. Il valore di ritorno avrà lo stesso tipo della base.

Ritorna

Il valore di ritorno avrà lo stesso tipo di *variable*.

Semantica

Il valore di ritorno della funzione avrà molto spesso il tipo sbagliato. Per il momento è consigliabile convertire la base ad un tipo più congeniale prima di eseguire la funzione.

Esempi

```
1. c:= 3
2. e:= 3.0
3. r:= sqrt(c) //r=1
4. r:= sqrt(e) //r= 1.7320507764816284
5. r:= nroot(c, 3) //r=?
6. r:= nroot(e, 3) //r=?
```

mod

Descrizione

Calcola il modulo.

Sintassi

mod(*variable,mod***)**

Parametri

- *variable* scalare o letterale numerico.
- *mod* letterale numerico, indica il modulo.

Restrizioni

Entrambi i parametri devono essere valori numerici. Il valore di ritorno avrà lo stesso tipo della base

Ritorna

Il valore di ritorno avrà lo stesso tipo di *variable*.

Semantica

Il valore di ritorno della funzione avrà molto spesso il tipo sbagliato. Per il momento è consigliabile convertire la base ad un tipo più congeniale prima di eseguire la funzione.

Esempi

```
1. d:= 10
2. r:= mod(d, 3) //r=1
```

Operatori e funzioni Booleani

=, >, <, <=, >=, <>

Descrizione

Verificano la relazione che intercorre tra due valori.

Sintassi

ds1 [**=** | **>** | **<** | **<=** | **>=** | **<>**] *ds2*

ds ::= [*variable* | *literal*]

Parametri

- *ds* scalare o letterale, i tipi supportati variano in base all'operazione:
  - **=, <>**: numerici, stringhe, booleani, date
  - **<,>,<=,>=** : numerici, stringhe, date.

Restrizioni

I valori booleani hanno un supporto parziale. Infatti, solo uguaglianza e disuguaglianza possono verificare valori booleani, gli altri no. La cosa ha senso: è possibile verificare se **true=true** o **false<>true** ad es., ma non è possibile stabilire se **true>false** (ad es.).

In teoria, anche i casi **<= e >=** potrebbero accettare booleani (tornerebbero **true** se e solo se i valori sono uguali, **false** altrimenti), nella pratica la cosa non è accettata.

Inoltre, entrambi i valori devono avere lo stesso tipo, in caso contrario viene lanciata un'eccezione.

Attualmente non supportano DataSet e Colonne.

Semantica

La valutazione avviene in modo diverso in base al tipo di dati: per i **numerici** ci si affida, banalmente, all'ordine, indipendentemente dal fatto che siano interi o float. Per le **stringhe**, si fa uso dell'ordine lessicografico. Per le date, si usano funzioni interne delle classi **Java.time**.

Ritorna

Uno scalare booleano con il risultato della valutazione.

Esempi

```
1. x:= ”giovanni”
2. y:= 3.14
3. r:= x <> ”Giovanni” //r=true
4. r:= y >= 3 //r=true
```

in, not in

Descrizione

Verificano se un valore appartiene ad una lista.

Sintassi

variable {**not**}? **in(** *literal {, literal}** **)**

Parametri

- *variable* scalare o letterale, numerico o stringa.
- **not** parametro opzionale che nega il risultato di **in().**
- *literal* lista di letterali (numerici o stringhe).

Restrizioni

Attualmente *variable* non accetta DataSet o colonne.

La lista di letterali deve essere dello stesso tipo. Non è possibile, ad es., mischiare stringhe con valori numerici. Inoltre, non vengono accettati booleani (sarebbe inutile) o date (verranno aggiunte in futuro).

Ritorna

Uno scalare booleano con il risultato della valutazione.

Semantica

Per un problema di grammatica, non è stato possibile negare il risultato di questa funzione tramite operatore unario **not**, dato che per le specifiche **not** andrebbe posto vicino alla parola chiave **in**, ma essendo un operatore unario, la sua posizione è alla sinistra dell'operatore, quindi:

```
1. a:= "pippo"
2. r:= a not in ("pippo", "pluto", "paperino") // r = false
```

dovrebbe essere invece, date le specifiche:

```
1. a:= "pippo"
2. r:= not(a in ("pippo", "pluto", "paperino")) // r = true
```

*le parentesi intorno a **in** sono mie, per evidenziare che **not** è un operatore separato.*

Che è equivalente alla scrittura estesa:

```
1. a:= "pippo"
2. r:= a in ("pippo", "pluto", "paperino")    // r = false
3. r:= not r                                  // r = true
```

ciò non toglie che sia possibile usare entrambi i modi insieme, ma a questo punto si negherebbe la negazione.

Esempi

```
1. X:= "IT"
2. ret:= X in ("UK", "ES", "DE", "FR") //ret=false
```

isnull, not isnull

Descrizione

Verifica se un valore è nullo oppure no.

Sintassi

variable {**not**}? **isnull(** *expression* **)**

Parametri

- *variable* scalare di qualsiasi tipo.
- **not** parametro opzionale che nega il risultato della funzione.
- *expression* un'espressione che, una volta valutata, ritorna uno Scalare.

Restrizioni

Attualmente non supporta DataSet o Colonne.

Ritorna

Uno scalare booleano con il risultato della valutazione.

Semantica

Non ci sono restrizioni sul tipo di scalare valutabile perché ogni scalare ha un valore nullo. O, perlomeno, è possibile creare uno scalare di un tipo specifico che sia nullo. Anche in questo caso fare riferimento alla documentazione del codice per ulteriori informazioni sui valori nulli.

Per quanto riguarda il parametro **not**, si faccia riferimento al paragrafo *semantica* dell'istruzione **in**, dato che si tratta esattamente della stessa situazione.

Esempi

```
1. X:= null
2. r:= X not isnull(X) //r=false
```

and, or

Descrizione

Operatori per calcolare **and,or** logico.

Sintassi

expression [**and** | **or**] *expression*

Parametri

- *expression* un'espressione che, alla fine della sua valutazione, ritorna uno scalare di tipo booleano o null.

Restrizioni

Non è possibile effettuare una valutazione su valori diversi da booleani e null.

Attualmente non supporta DataSet o Colonne

Ritorna

Uno scalare booleano con il risultato della valutazione.

Semantica

Questi operatori si basano su una particolare tavola della verità che tiene conto anche di valori **nulli**. In generale, infatti, questi vengono valutati come se fossero **false**, ma spesso tornano come risultato **null.**

Per un'indicazione più precisa, di seguito è indicata la tavola della verità completa presa dal manuale utente VTL:

X	Y	not X	X and Y	X or Y
T	T	F	T	T
T	F	F	F	T
T	*N*	F	*N*	T
F	T	T	F	T
F	F	T	F	F
F	*N*	T	F	*N*
N	T	*N*	*N*	T
N	F	*N*	F	*N*
N	*N*	*N*	*N*	*N*

Legenda: T *true*, F *false*, N *null*

Generalmente questi operatori sono usati per concatenare o comunque valutare una serie di condizioni diverse, laddove è possibile inserirne una sola: come ad esempio nei RuleSet, che considerano solo due condizioni (antecedente e conseguente) ma che con **and/or** sono espandibili anche a più di due condizioni.

Esempi

```
1. NAME:= ”pippo”
2. AGE:= 22
3. r:= (NAME=”peppe”) or (AGE>=18) //r=true
```

Operatori e Funzioni di validazione

check(DataPoint RuleSet)

Descrizione

Questa istruzione applica uno o più RuleSet ad ogni DataPoint di un DataSet.

Sintassi

check(
ds,
{dprs}+
{, [**not valid** | **valid** | **all**]*}?*
{, [**condition** | **measures**]*}?*
)

Parametri

- *ds* il DataSet a cui si deve applicare le regole.
- *dprs* una lista di RuleSet che verranno valutati tutti per ogni DataPoint.
- **not valid** | **valid** | **all** : opzionale, indica quali DataPoint verranno mantenuti in base al risultato della valutazione delle regole. Se non indicato, si assume **not valid**.
  condition | **measures** : opzionale, indica come indicare nella tabella il risultato della valutazione. Se non indicato, si assume **condition.**

Restrizioni

I RuleSet devono essere applicati al DataSet specificato: significa che le colonne indicate nei RuleSet devono essere presenti nel DataStructure del DataSet.

Obbligatoriamente devono essere presenti un solo DataSet ed almeno uno RuleSet.

Non è possibile usare l'opzione **all** con **measures**. Il motivo è che la valutazione ritornerebbe il DataSet originario.

Se si specifica un parametro, è necessario specificare anche l'altro. Non è possibile indicare solo uno dei parametri opzionali, anche se in teoria per l'altro si dovrebbe assumere il valore di default.

I parametri opzionali vanno ovviamente indicati nell'ordine corretto.

Ritorna

Se usato il parametro **measures**, un DataSet con tutte le colonne di *ds*, ma con i soli DataPoint che sono stati validati dai RuleSet (se **valid**) oppure i DataPoint che hanno fallito la verifica dei RuleSet (se **not valid**).

Se usato il parametro **condition**, un DataSet con tutte le colonne di *ds*, con l'aggiunta di una colonna **CONDITION** che contiene un booleano che esprime il risultato della valutazione.

In entrambi i casi, ulteriori colonne vengono aggiunte:

- **ERRORMESSAGE**, che contiene il codice/messaggio d'errore specificato dal parametro **errorcode** nel RuleSet.
- **RULE_ID**, contiene la concatenazione dell'ID del RuleSet e l'ID della regola (interna a quel RuleSet) che ha fallito nel verificare la regola. La concatenazione avviene tramite *underscore*.

Semantica

Per quanto riguarda il messaggio/codice di errore, questa colonna viene popolata solo se una regola ha fallito, mentre negli altri casi rimarrà vuota.

Ovviamente questa colonna non sarà mai riempita nel caso si scelga **valid**, sarà sempre piena quando si sceglie **not valid**, mentre può alternare caselle vuote e piene nel caso di **all**.

Il motivo per cui non si può usare **all** con **measures** è adesso evidente: ritornerebbe il dataset iniziale con la colonna **ERRORMESSAGE** in più.

Sfortunatamente non è stata prevista un'opzione che permetta di aggiungere solo la colonna **CONDITION**, senza i messaggi di errore.

Un'ulteriore precisazione è importante per quanto riguarda **ERRORMESSAGE**: nelle regole non è obbligatorio specificarlo. Questo significa che la colonna **ERRORMESSAGE** potrebbe risultare vuota, anche se un DataPoint ha fallito nella verifica contro quella regola.

Nelle specifiche non viene poi affrontato un problema importante: cosa fare quando un DataPoint fallisce più di una regola di più di un RuleSet e quindi cosa inserire nella colonna **ERRORMESSAGE**. La soluzione scelta è quello di inserire il primo messaggio di fallimento relativa alla prima regola del primo RuleSet fallito.

Esempi

ds1		
ID	**NAME**	**AGE**
0	Marco	36
1	Luigi	12
2	Peppe	30

```
define datapoint ruleset rs1(NAME, AGE) {
    RL11: when NAME = "peppe" then not isnull(AGE) errorcode("regola 11 fallita")
}
define datapoint ruleset rs2(AGE){
    RL12: when true then (AGE>18 and AGE<35) errorcode("regola 12 fallita")
}
chk1:= check(ds1, rs1, valid, condition)
chk2:= check(ds1, rs2, not valid, measure)
```

Il significato della regola *rs1* è: **quando** NAME è uguale a "peppe" **quindi** il campo AGE non è nullo, la regola è valida.

Di conseguenza, esaminando il DataSet *ds1*, è evidente che c'è un solo DataPoint che verifica questa regola. La funzione **check** ha parametri (**valid,condition**), quindi ci aspettiamo di avere come risultato tutti i campi che rispettano questa regola, più una colonna che ci informa del risultato della validazione. Otteniamo quindi come risultato per **chk1**:

chk1				
ID	**NAME**	**AGE**	**CONDITION**	**ERRORMESSAGE**
2	Peppe	30	true	

Si noti che il campo ERRORMESSAGE è vuoto, con valore Stringa nullo.

Il significato della regola *rs2* è: (*ignora antecendete*) **quindi** il campo AGE è maggiore di 18 **e** il campo AGE è minore di 35, la regola è valida.

La funzione **check** ha parametri (**not valid,measures**), questo significa che avremo solo la colonna **ERRORMESSAGE** in più nel DataStructure. Tuttavia, avendo il parametro **not valid**, tutte le righe di questa colonna saranno non vuote (dato che **errorcode** è settato per la regola *rs2*).

Otteniamo, per **chk2**:

chk2			
ID	**NAME**	**AGE**	**ERRORMESSAGE**
0	Marco	36	regola 12 fallita
1	Luigi	12	regola 12 fallita

Funzioni su Insiemi

union

Descrizione

Unisce due o più DataSet senza duplicati.

Sintassi

union(*ds {, ds}**)**

Parametri

- *ds* DataSet

Restrizioni

Tutti i DataSet devono obbligatoriamente avere i componenti *Identifier* e *Measure* uguali (stesso nome e tipo). I campi *Attribute* vengono ignorati.

Se anche solo uno dei DataSet ha struttura diversa, verrà lanciata un'eccezione.

Ritorna

Un DataSet con i componenti *Identifier* e *Measure* e tutti i DataPoint unici (non duplicati) dei DataSet passati come parametro.

Semantica

Come da specifiche VTL, passare un solo DataSet non genera problemi né a livello di grammatica né a livello di esecuzione. Semplicemente verrà ritornato quel DataSet senza alcuna modifica.

Esempi

ds1		
ID	**NAME**	**AGE**
0	Marco	36
1	Luigi	12
2	Peppe	30

ds2		
ID	**NAME**	**AGE**
1	Luigi	12
2	Peppe	30
3	Antonio	24

```
1. dsr:= union(ds1, ds2)
```

dsr		
ID	**NAME**	**AGE**
0	Marco	36
1	Luigi	12
3	Antonio	24

intersect

Descrizione

Ritorna l'intersezione di due o più DataSet.

Sintassi

intersect(*ds {, ds}** **)**

Parametri

- *ds* DataSet

Restrizioni

Tutti i DataSet devono obbligatoriamente avere i componenti *Identifier* e *Measure* uguali (stesso nome e tipo). I campi *Attribute* vengono ignorati.

Se anche solo uno dei DataSet ha struttura diversa, verrà lanciata un'eccezione.

Ritorna

Un DataSet con i componenti *Identifier* e *Measure* e tutti i DataPoint in comune dei DataSet passati come parametro.

Semantica

Anche se non chiaramente indicato dalle specifiche VTL, passare un solo DataSet non genera problemi né a livello di grammatica né a livello di esecuzione. Semplicemente verrà ritornato quel DataSet senza alcuna modifica. Si noti la disparità tra la funzione **union** e questa nelle specifiche originali VTL.

Un DataPoint viene considerato in comune quando presente in tutti i DataSet indicati. Ovviamente si ignorano eventuali campi attributo.

Esempi

ds1		
ID	**NAME**	**AGE**
0	Marco	36
1	Luigi	12
2	Peppe	30

ds2		
ID	**NAME**	**AGE**
0	Marco	36
3	Pietro	21
4	Luca	23

```
1. dsr:= intersect(ds1, ds2)
```

dsr		
ID	**NAME**	**AGE**
0	Marco	36

symdiff

Descrizione

Calcola la differenza simmetrica di due DataSet.

Sintassi

symdiff(*ds1 , ds2* **)**

Parametri

- *ds1,ds2* DataSet

Restrizioni

I due DataSet devono obbligatoriamente avere i componenti *Identifier* e *Measure* uguali (stesso nome e tipo). I campi *Attribute* vengono ignorati.

Se i DataSet hanno struttura diversa, verrà lanciata un'eccezione.

Ritorna

Un DataSet con i componenti *Identifier* e *Measure* e tutti i DataPoint che esistono in uno ma non nell'altro DataSet passati come parametro.

Semantica

Si noti che questa funzione è l'inverso di **intersect**, infatti restituisce esattamente i DataPoint che **intersect** scarterebbe. Ovviamente **setdiff** lavora su solo due DataSet per volta, a differenza di **union** e **intersect.**

Esempi

ds1		
ID	**NAME**	**AGE**
0	Marco	36
1	Luigi	12
2	Peppe	30

ds2		
ID	**NAME**	**AGE**
0	Marco	36
3	Pietro	21
4	Luca	23

```
1. dsr:= symdiff(ds1, ds2)
```

dsr		
ID	**NAME**	**AGE**
1	Luigi	12
2	Peppe	30
3	Pietro	21
4	Luca	23

setdiff

Descrizione

Calcola la differenza di due DataSet.

Sintassi

setdiff(*ds1 , ds2* **)**

Parametri

- *ds1,ds2* DataSet

Restrizioni

I due DataSet devono obbligatoriamente avere i componenti *Identifier* e *Measure* uguali (stesso nome e tipo). I campi *Attribute* vengono ignorati.

Se i DataSet hanno struttura diversa, verrà lanciata un'eccezione.

Ritorna

Un DataSet con i componenti *Identifier* e *Measure* e tutti i DataPoint che esistono in *ds1* ma non in *ds2*.

Semantica

Si differenzia da **symdiff** per il semplice fatto che **setdiff** non controlla se ci sono elementi in *ds2* che non esistono in *ds1*. Risulta utile quando vogliamo filtrare i DataPoint di un DataSet, avendo a disposizione già i DataPoint da scartare, quindi senza dover ricorrere a **filter**. Cosi come **symdiff**, anche questa lavora su due DataSet.

Esempi

ds1		
ID	**NAME**	**AGE**
0	Marco	36
1	Luigi	12
2	Peppe	30

ds2		
ID	**NAME**	**AGE**
0	Marco	36
3	Pietro	21
4	Luca	23

```
1.  dsr:= setdiff(ds1, ds2)
```

dsr		
ID	**NAME**	**AGE**
1	Luigi	12
2	Peppe	30

Funzioni Condizionali

nvl

Descrizione

Sostituisce i valori nulli con lo Scalare indicato.

Sintassi

nvl(*ds , expression* **)**

Parametri

- *ds* DataSet
- *expression* espressione che, una volta valutata, restituisce uno scalare, oppure uno Scalare o un letterale.

Restrizioni

Il DataSet deve avere tutti i campi *measure* dello stesso tipo dello Scalare ritornato da *expression*, altrimenti viene lanciata un'eccezione.

Ritorna

Un DataSet i cui campi *measure* sono tutti non nulli.

Semantica

Questa istruzione è estremamente specifica, anche per le specifiche VTL. Non funziona se il DataSet indicato ha dei campi *measure* di tipo diverso. Molto più intelligente sarebbe stato poter usare colonne di un DataSet, per poter usare praticamente tutti i DataSet.

Esempi

ds1		
ID *(identifier)*	**NAME** *(identifier)*	**AGE** *(measure)*
0	Marco	36
1	Luigi	
2	Peppe	30

```
1.  dsr:= nvl(ds1, 12)
```

ds1		
ID *(identifier)*	**NAME** *(identifier)*	**AGE** *(measure)*
0	Marco	36
1	Luigi	12
2	Peppe	30

If/then/else

Descrizione

Valuta una condizione e decide quale valore ritornare dalle alternative.

Sintassi

If *expression* **then** *expression* {**elseif** *expression* **then** *expression*}* **else** *expression*

Parametri

- *expression* espressione che, una volta valutata, restituisce uno scalare booleano (per **if/elseif**) o un DataSet (per **then/else**).

Restrizioni

Expression deve tornare obbligatoriamente uno Scalare Booleano per i token **if** e **elseif**, mentre per **then/else** si dovrebbe tornare un DataSet, ma non viene fatto un controllo, perché viene demandato all'istruzione che contiene questa funzione (vedi esempio).

Come da sintassi, per ogni **elseif** deve essere presente il rispettivo **then**, pena il fallimento del parsing.

Ritorna

Un VTLObj qualsiasi, ma nella pratica DataSet o Scalare (qualsiasi).

Semantica

A differenza di praticamente tutti i linguaggi di programmazione, l'**if/the/else** di VTL funziona solo *in-line* in una assegnazione o come parametro di una funzione per scegliere quale DataSet usare.

Attenzione: siccome non è stato previsto un controllo sui dati di output (quindi se è un DataSet), si potrebbe pensare di usare questa funzione anche per tipi di dato diversi di DataSet. L'operatore/Funzione che usa questa istruzione deve controllare accuratamente il tipo di dato ritornato.

Esempi

```
1. X:= ”giovanni”
2. Y:= ”antonio”
3. Z:= if (X <> Y) then “T” else “F”
4. //Z = “T”
```

Join

Queste particolari istruzioni permettono di effettuare la join tra DataSet e consequenzialmente applicare una o più clausole al DataSet risultante. Sono le istruzioni più complicate di tutto il set.

Inner

Descrizione

Unisce tutte le colonne di un DataSet con le colonne dell'altro, considerando solo le righe che verificano una certa condizione.

Sintassi

[inner *ds1* **as** *alias*, *ds2* **as** *alias* **on** *condOp* **] {** *clauseBody+* **}**

*condOp ::= ds1.k [***=** | **<** | **>** | **<>** | **<=** | **>=***] ds2.k*

clauseBody ::= [**keep** | **rename** | **drop** | **filter** | **calc** *]*

Parametri

- *ds1, ds2* DataSet su cui verrà effettuata l'operazione.
- *alias* un *stringLiteral* che indica il nome temporaneo che verrà usato nell'elaborazione delle clausole e delle condizioni di join.
- *condOp* la condizione che deve essere valutata per verificare se un datapoint può essere aggiunto. Le espressioni condiziali usabili non si limitano a quelle indicate. Una lista più completa è in divenire.
- *clauseBody* le varie clausole (almeno una) che verranno valutate dopo la join.

Restrizioni

alias deve essere sempre specificato.

condOp deve obbligatoriamente essere una espressione condizionale che quindi deve ritornare uno Scalare booleano alla fine della sua valutazione, per ogni DataPoint.

Nelle *clauseBody*, le variabili devono obbligatoriamente essere espresse come appartenenza, quindi nella forma *ds.k* o, in questo caso, *alias.k*. Questo perché non è possibile sapere con precisione a quale colonna si fa riferimento, avendo la join entrambe le colonne dei DataSet.

Deve essere presente almeno una clausola.

Ritorna

Un DataSet la cui struttura dipende dalle operazioni effettuate.

Semantica

Questa operazione è peculiare: cerca di mimare quello che in SQL viene fatto più banalmente tramite INNER JOIN di SQL. In quest'ultimo, le colonne da mantenere vengono subito indicate, mentre le condizion vengono poste alla fine. Qui invece è tutto al contrario, e costringe ad aggiungere clausole per poter filtrare righe/colonne prima di ritornare un risultato.

Per essere più chiari, facciamo un esempio e paragoniamo join SQL e VTL:

WORKER		
ID	**NAME**	**AGE**
0	Marco	36
1	Luigi	12
2	Peppe	30

JOBS		
WORKERID	**POSITION**	**SALARY**
0	IT	1000
1	CEO	2000
2	HR	1500

Ipotizziamo vogliamo la lista dei nomi dei lavoratori con la rispettiva posizione in azienda.
In SQL scriveremmo

```
1.  select WORKER.NAME, JOBS.POSITION
2.         from WORKER, JOBS
3.         where WORKER.ID = JOBS.WORKERID
```

Si noti che si è usata la forma equivalente della *inner join*, dove non è necessario indicarla esplicitamente (vale solo per la INNER, per le altre bisogna indicarle). Otteniamo quindi:

RESULT	
NAME	**POSITION**
0	IT
1	CEO
2	HR

NB: il nome della tabella è esemplificativo e dipende dal DBMS usato.

In VTL, per ottenere lo stesso risultato, dobbiamo invece scrivere:

```
RESULT: = [inner WORKER as “A”, JOBS as “B” on A.ID = B.WORKERID] {
    keep(A.NAME, B.POSITION)
}
```

Che ritorna la stessa tabella di prima, con l’unica vera differenza che adesso la tabella risultate si chiama proprio RESULT.

Vediamo quindi che la *SELECT* è la **keep**, *FROM* è (**WORKER as “A”, JOBS as “B”)** mentre la *WHERE* è la condizione espressa dalla **on**.

Seppur meno *user-friendly*, la versione VTL fornisce un vantaggio in termini di manipolazione del risultato: possiamo infatti filtrare ulteriormente i DataPoint, rinominare una o più colonne o addirittura ricalcolare i valori dei componenti prima di ritornare un risultato.

In SQL tutto ciò ci avrebbe portato ad una serie di ALTER/UPDATE seguenti la JOIN.

Esempi

*Vedi **Semantica** per un esempio di applicazione.*

Funzioni analitiche

Funzioni di aggregazione

Descrizione

Queste funzioni vengono usate per elaborare dati statistici e aggregare valori.

Sintassi

*aggregateFun***(***ds, {param}?***) group by(***idcomp (, idcomp)****)**

Parametri

- *aggregatFun* la funzione di aggregazione scelta. La tabella con le funzioni è presente nel paragrafo *Semantica*.
- *param* è un parametro aggiuntivo, che deve essere indicato solo se la funzione di aggregazione usata lo richiede. Con le funzioni attualmente implementate, questo parametro non sarà mai richiesto.
- *ds* può essere un DataSet o una colonna.
- *idcomp* è il nome del componente **Identifier** del DataSet.

Restrizioni

ds deve contenere esattamente una colonna con ruolo **Measure**, di tipo numerico. Nel caso in cui si indica una colonna, questa deve essere un **Measure**, ma non vale più la restrizione precedente.

idcomp ovviamente deve esistere nel DataSet e deve essere obbligatoriamente di tipo **Identifier**.

Ritorna

Un DataSet con i campi **Identifier** specificati da *idcomp*, e il campo **Measure** di *ds* (o indicato tramite appartenenza). Il numero di righe dipende dal tipo di operazione di aggregazione e dal valore delle variabili.

Semantica

Le funzioni di aggregazione attualmente implementate sono le seguenti:

Funzione	Descrizione
max(*ds*)	Ritorna il massimo valore per gruppo.
min(*ds*)	Ritorna il valore minimo per gruppo.
sum(*ds*)	Ritorna la somma di tutti i valori del gruppo.
avg(*ds*)	Ritorna una media aritmetica di tutti i valori del gruppo.

La funzione di aggregazione **avg** viene calcolata passo dopo passo, tramite una formula derivata da quella della media aritmetica, in modo che sia possibile calcolare la media in passi successivi, invece che dover sommare tutti i valori (con possibile **BufferOverflow**) tutto alla fine.

Partendo quindi dalla formula della media aritmetica:

$$u = \frac{\sum_{i=1}^{n} x_i}{n}$$

Si deriva la seguente formula:

$$u_n = \frac{(n-1) * u_{n-1} + x_n}{n}$$

Che calcola una nuova media a partire dalla precedente.

Importante notare che i DataPoint il cui campo Measure sia nullo verranno scartati, in quanto non portano alcun contributo statistico.

Potrebbe risultare problematico nel caso in cui un DataPoint, con campo Measure nullo, sia anche unico a seguito dell'operazione di aggregazione, perché questo verrebbe scartato, provocando una possibile perdita di informazioni.

La funzione di aggregazione non viene applicata se un DataPoint è unico.

Esempi

ds1		
ID *(identifier)*	**NAME** *(identifier)*	**AGE** *(measure)*
0	Marco	36
1	Luigi	12
2	Peppe	30
3	Marco	24

```
1. dsr:= max(ds1) group by(NAME)
```

dsr	
NAME *(identifier)*	**AGE** *(measure)*
Marco	36
Luigi	12
Peppe	30

```
1. dsr:= sum(ds1) group by(NAME)
```

dsr	
NAME *(identifier)*	**AGE** *(measure)*
Marco	60
Luigi	12
Peppe	30

```
1. dsr: = avg(ds1) group by(NAME)
```

dsr	
NAME *(identifier)*	**AGE** *(measure)*
Marco	30
Luigi	12
Peppe	30

Clausole

rename

Descrizione

Permette di cambiare il nome e/o il ruolo di uno o più componenti di un DataSet.

Sintassi

ds **[rename(** *renameBody {, renameBody}** **]**

renameBody ::= compName **as** *newCompName {* **role** *[* ***identifier*** *|* ***measure*** *|* ***attribute*** *] }?*

Parametri

- *ds* DataSet da modificare.
- *compName* nome del componente da rinominare.
- *newCompName* il nuovo nome del componente, espresso come letterale stringa.
- **role** opzionale, indica il nuovo ruolo che il componente deve assumere.

Restrizioni

Ovviamente i *compName* devono esistere all'interno del DataStructure di *ds*, in caso contrario viene lanciata un'eccezione.

Inoltre, *newCompName* deve essere un nome unico, per non sovrascrivere un componente già esistente.

Ritorna

Un nuovo DataSet con il DataStructure modificato.

Semantica

Questa clausola è molto simile a **keep** (infatti condividono buona parte del codice), con l'unica differenza che keep è distruttiva, mentre rename mantiene quanto più possibile i dati presenti nel Data Set obiettivo.

Esempi

ds1		
ID *(identifier)*	**NAME** *(identifier)*	**AGE** *(measure)*
0	Marco	36
1	Luigi	12
2	Peppe	30

```
1. ds2:= ds1[rename(AGE as “WEIGHT” role attribute, NAME role measure)]
```

Questo comando rinomina il campo AGE come WEIGHT con ruolo *attribute* e cambia il ruolo del campo NAME facendolo diventare *measure*. Il nuovo DataSet sarà così composto:

ds2		
ID *(identifier)*	**NAME** *(measure)*	**WEIGHT** *(attribute)*
0	Marco	36
1	Luigi	12
2	Peppe	30

Lasciando inalterati i DataPoint.

keep

Descrizione

Permette di mantenere solo alcune componenti di un DataSet, cambiandone nome e/o ruolo se necessario.

Sintassi

ds **[keep(** *keepBody {, keepBody }** **]**

keepBody ::= compName { **as** *newCompName}? {* **role** *[* ***identifier*** *|* ***measure*** *|* ***attribute*** *] }?*

Parametri

- *ds* DataSet da modificare.
- *compName* nome del componente da mantenere.
- *newCompName* opzionale, il nuovo nome del componente, espresso come letterale stringa.
- **role** opzionale, indica il nuovo ruolo che il componente deve assumere.

Restrizioni

Gli stessi di **rename**.

Ritorna

Un nuovo DataSet con il DataStructure modificato.

Semantica

Nella scrittura di uno script, si fa notare che non sempre è preferibile optare per **rename**, perché la **keep** può ridurre considerevolmente la quantità di dati da mantenere in memoria, velocizzando eventualmente il tempo di esecuzione delle operazioni che useranno il Data Set in output come input, facendo esattamente le stesse operazioni (ridenominazione e cambio ruolo).

as qui diventa opzionale. Questo significa che è possibile elencare solamente le colonne che si vogliono mantenete.

Esempi

ds1		
ID *(identifier)*	**NAME** *(identifier)*	**AGE** *(measure)*
0	Marco	36
1	Luigi	12
2	Peppe	30

```
1. ds2:= ds1[keep(ID, NAME role measure)]
```

Questa istruzione mantiene solo i componenti **ID,NAME**, modificando il ruolo del secondo a *measure*.

ds2	
ID *(identifier)*	**NAME** *(measure)*
0	Marco
1	Luigi
2	Peppe

filter

Descrizione

Crea un nuovo DataSet aventi solo i DataPoint che rispettano una condizione indicata come parametro.

Sintassi

ds **[filter(** *expression {, expression }** **]**

Parametri

- *ds* DataSet da modificare.
- *expression* espressione condizionale da valutare.

Restrizioni

L'espressione deve ritornare uno scalare booleano, indipendentemente dalle operazioni e dalla sua complessità.

Ritorna

Un DataSet con i soli DataPoint che verificano ognuna delle condizioni passate come parametro.

Semantica

Un'espressione può anche tornare *null*. In questo caso si agisce come se fosse *false* e si scarta il DataPoint corrente.

Esempi

ds1		
ID	**NAME**	**AGE**
0	Marco	36
1	Luigi	12

2	Peppe	30

```
1. ds2:= ds1[filter(NAME=”peppe”, AGE>18)]
```

ds2		
ID	**NAME**	**AGE**
2	Peppe	30

calc

Descrizione

Ricalcola i valori di uno o più componenti di un DataSet o aggiunge una nuova colonna con i valori calcolati.

Sintassi

ds **[calc(** *calcBody {, calcBody }** **]**

calcBody::= compValue **as** *CompName {* **role** *[* ***identifier*** *|* ***measure*** *|* ***attribute*** *] }?*

Parametri

- *ds* DataSet da modificare.
- *compValue* un’espressione che ritorna uno scalare.
- *CompName* il nome del componente a cui va ricalcolato il valore.
- **role** opzionale, indica il nuovo ruolo che il componente deve assumere.

Restrizioni

compValue deve restituire uno Scalare, tipo indifferente, dello stesso tipo per ogni DataPoint analizzato. L’unica variazione di tipo ammessa è **null**.

Ritorna

Un nuovo DataSet con il DataStructure/DataPoint modificato che ha cardinalità uguale a *ds.*

Semantica

Nel caso in cui *CompName* non faccia riferimento ad un componente del DataStructure, **calc** provvede a creare un nuovo componente con quel nome e il valore calcolato.

Per quanto riguarda il ruolo, l’unico caso degno di nota è quando *CompName* non esiste e non è stato specificato un ruolo. In questo caso, si assume che il componente abbia come ruolo *measure.* In tutti gli altri casi, o si usa il ruolo indicato (nel caso il componente esiste, si sovrascrive) oppure si eredita il ruolo del componente esistente.

Esempi

ds1		
ID *(identifier)*	**NAME** *(identifier)*	**AGE** *(measure)*
0	Marco	36
1	Luigi	12
2	Peppe	30

```
ds2:=  ds1[calc(
            (AGE+1) as AGE,
            (AGE*2) as WEIGHT role attribute
               )
          ]
```

Questa clausola modifica il valore del componente **AGE** aggiungendo 1 al valore precedente, con ruolo invariato. Aggiunge inoltre una nuova colonna con ruolo attributo e valore **AGE*2**.

Gli altri campi sono rimasti invariati.

ds2			
ID *(identifier)*	**NAME** *(identifier)*	**AGE** *(measure)*	**WEIGHT** *(attribute)*
0	Marco	37	52
1	Luigi	13	24
2	Peppe	31	60

drop

Descrizione

Inverso di **keep**, rimuove i componenti specificati dal DataStructure e ricalcola il DataSet associato.

Sintassi

ds **[drop(** *varname {, varname }** **]**

Restrizioni

Contrariamente a quanto si possa immaginare, *varname* non deve essere un nome di componente valido. Se esiste, viene scartato dal DataStructure, se invece non esiste, viene semplicemente ignorato.

Ovviamente, nel caso in cui si indichino tutte le colonne, e quindi si ottenga un DataSet senza alcun componente, verrà lanciata un'eccezione.

Ritorna

Il DataSet originario meno i componenti indicati.

Se vengono indicati solo componenti inesistenti (e quindi il DataStructure risultate ha lo stesso numero di componenti dell'originale), viene ritornato il DataSet originale senza ulteriori operazioni.

Semantica

Questa clausola è l'unica che non effettua molti controlli sui parametri in ingresso. Di conseguenza, è molto meno propensa a lanciare eccezioni o ad andare in errore. Nel caso in cui trova una situazione ambigua, ignora e passa oltre.

Ad esempio, se gli viene passato un nome di componente che non è presente nel DataStructure originario, e questo è l'unico parametro, allora dopo la verifica del DataStructure, l'istruzione si renderà conto che il numero di componenti è rimasto invariato, e ritornerà il DataSet originario.

Ovviamente bisogna prestare particolare attenzione all'uso di questa istruzione, perché **distruttiva** verso i dati.

Esempi

ds1		
ID *(identifier)*	**NAME** *(identifier)*	**AGE** *(measure)*
0	Marco	36
1	Luigi	12
2	Peppe	30

```
1. ds2:= ds1[ drop(NAME) ]
```

ds2	
ID *(identifier)*	**AGE** ***(measure)***
0	36
1	12
2	30

Named Functions/Procedures

In questo capitolo si descrive il meccanismo di definizione e uso di funzioni e procedure.
Bisogna considerarlo come un'appendice, quindi la struttura varierà leggermente dal resto del documento.

Come in molti altri linguaggi, anche in VTL è possibile definire procedure e funzioni e usarle all'interno di uno script. La cosa importante da notare, però, è che in VTL queste vengono intese in un modo leggermente differente rispetto agli altri linguaggi.

La fase di definizione e il meccanismo di chiamata risultano anch'essi leggermente differenti rispetto al solito.

Per quanto riguarda la definizione, infatti, le procedure fanno parte del gruppo define (*vedi pag. 6)*, in quanto condividono la stessa parola chiave (*define*, appunto). Al contrario, le funzioni fanno uso della parola chiave **create**.

La differenza probabilmente risiede nel loro scopo ultimo: le procedure sono composte da una sequenza più o meno lunga di istruzioni (generalmente assegnazioni), con un valore in input che funge da output, assolvendo in questo modo ad uno specifico *task*. Le funzioni invece sono composte da un'unica istruzione (**returns**) più o meno complessa, il cui scopo è computare il valore di ritorno.

Vale la pena di notare che anche l'uso differisce: le prime vanno usate da sole, mentre le seconde vanno obbligatoriamente usate in un'assegnazione, cosa che permette di concatenarle in una espressione più complessa, se necessario.

Così come per i RuleSet, anche qui vige la regola che una funzione/procedura per essere chiamata, deve essere stata prima definita.

Si esamina quindi prima il meccanismo di definizione per entrambe e successivamente il meccanismo di richiamo.

define procedure

Descrizione

Permette di definire una procedura.

Sintassi

define procedure *varname***(** *inputparam+* **,***outputparam* **)**{

assignment+

}

inputparam ::= **input** *varname* **as** [**dataset**|**string**]

outputparam ::= **output** *varname* **as** [**dataset**|**string**]

Parametri

- *varname* è il nome che identifica univocamente una procedura.
- *inputparam* è una lista (almeno uno) di parametri di input. Questi possono essere o DataSet o String, seppur in questo caso non parliamo di LiteralString, bensì di *varname* che possono essere usati indifferentemente come Scalar, come riferimento ad un RuleSet o come riferimento a colonne. In quest'ultimo caso ovviamente deve essere presente almeno un altro parametro di tipo DataSet.
- *outputparam* è un parametro (unico) che definisce la variabile che verrà restituita in output alla fine dell'elaborazione della procedura.
- *assignment* indica semplicemente che all'interno delle procedure è possibile usare tutti quei operatori e quelle funzioni VTL che ritornano un valore, che deve essere assegnato ad una variabile.

Restrizioni

Partendo dai parametri, è importante notare che non viene effettuato un controllo sul nome della procedura: questo implica che se si definiscono due procedure con lo stesso nome, la seconda andrà a sovrascrivere la prima in modo trasparente.

Una procedura deve avere obbligatoriamente almeno un parametro di input e uno di output, anche se i dati usati vengono generati internamente oppure l'elaborazione non restituisce alcun valore (*si fa notare che in quest'ultimo caso la creazione di una procedura diventa un'inutile esercizio di stile*).

Il parametro di output è unico e deve sempre essere definito come ultimo parametro. Questa restrizione riduce le potenzialità dello strumento, ma è temporanea e verrà modificata in una successiva versione.

Le istruzioni interne alla procedura devono necessariamente essere del tipo *var:=expr*, quindi devono tornare un valore. Non è quindi possibile usare istruzioni come **put**, **procedure, define** ed eventuali istruzioni di debug.

Ritorna

Nulla, seppur a livello di codice venga creato un riferimento nella memoria a questa procedura, pronta per essere richiamata.

Semantica

Come riportato nell'ultimo paragrafo della sezione relative alle restrizioni, una procedura può essere virtualmente considerata come uno script VTL eseguito all'interno di un altro script VTL.

Alcune istruzioni (ad es. **join**, **clausole**) vengono messe a conoscenza (tramite il flag **FLG_PRCDR_BODY**) del fatto di essere eseguite all'interno dello *scope* di una procedura, modificando di fatto il loro comportamento, leggendo le variabili passategli in modo differente.

L'uso tipico (e consigliato) di questo oggetto è quello di ridurre la ridondanza del codice, sostituendo con dei riferimenti tutti quelle istruzioni duplicate. Ad es. se dobbiamo effettuare una validazione/trasformazione di tutti i DataSet letti tramite **get**, risulta comodo riunire queste istruzioni in una procedura e richiamarla quando necessario, quindi da qualcosa come:

```
1.  define datapoint ruleset RLST01(CLMN_A, CLMN_B) {
2.      RL1: …
3.      RL2: …
4.  }
5.  ds1:= get("db/table1")
6.  dsr:= check(ds1, RLST01)
7.  dsr:= dsr[drop(ERRORMESSAGE)]
8.  ds1:= dsr
9.  ds2:= get("db/table2")
10. dsr:= check(ds1, RLST01)
11. dsr:= dsr[drop(ERRORMESSAGE)]
12. ds2:= dsr
13. …
```

avremo qualcosa del tipo:

```
1.  define datapoint ruleset RLST01(CLMN_A, CLMN_B) {
2.      RL1: …
3.      RL2: …
4.  }
5.  define procedure PR01(input DS as dataset, input RL as string, output RLST as dataset) {
6.      RLST:= check(DS, RL)
7.      RLST:= RLST[drop(ERRORMESSAGE)]
8.  }
9.  ds1:= get("db/table1")
10. PR01(ds1, RLST01, ds1)
11. ds2:= get("db/table2")
12. PR01(ds2, RLST01, ds2)
13. …
```

Una piccola nota su questo secondo listato: la chiamata a funzione del rigo 10 (ma vale lo stesso per la successiva al rigo 12) scritta in quel modo, cioè ponendo come input e come output contemporaneamente lo stesso DataSet, funziona perfettamente. Questo perché l'output viene traslato alla variabile passata come parametro solo alla fine dell'esecuzione, non durante. Ulteriori informazioni più avanti, alla chiamata della procedura.

Come si può vedere, nonostante i due listati abbiano lo stesso numero di righe totali, l'uso della procedura ha comunque dimezzato il numero di righe del corpo dello script, passando da 8 (*righe da 5 a 12*) a 4 (*righe da 9 a 12*), rendendolo contemporaneamente più leggibile e più facile da scrivere. Se avessimo aggiunto ulteriori letture, o avuto necessità di una più complicata validazione dell'input (in questo esempio), i benefici sarebbero diventati ancora più palesi.

Esempi

*Vedi il secondo listato nel paragrafo **Semantica.***

create function

Descrizione

Definisce una funzione.

Sintassi

create function *varname***(***varname+***){**

    returns *expr* **as** *DataType*

}

*Attenzione alla parola chiave **returns**, può capitare di confondersi e omettere la '**s**' finale, dato che in tutti gli altri linguaggi si usa al singolare.*

Parametri

- *varname* il primo identifica univocamente l'oggetto funzione, mentre tutti i successivi indicano i parametri da passare alla funzione.
- *expr* una espressione che ritorna un oggetto del tipo indicato da *DataType*
- *DataType* un qualsiasi tipo di dato supportato da VTL, più DataSet (integer, float, boolean, string, date, dataset).

Restrizioni

Cosi come per le procedure, anche qui se si usa un nome già usato per indicare la funzione, l'ultima andrà a sovrascrivere la precedente durante l'esecuzione.

Una funzione necessita di almeno un parametro per essere considerata valida: anche in questo caso questa è una limitazione temporanea. Presto sarà possibile creare funzioni senza parametro.

L'espressione può essere, come no, racchiusa da parentesi tonde. Le parantesi infatti agiscono come un operatore, seppur non eseguendo alcuna operazione e limitandosi a ritornare il valore in esse contenuto (leggasi il paragrafo relativo, pag. 12/13).

Non è possibile usare procedure o definire oggetti all'interno di una funzione, ma è possibile usare altre funzioni (*in teoria*) come parte di una espressione più complessa.

Ritorna

Un oggetto dello stesso tipo definito da *DataType*.

Semantica

Anche se a prima vista può sembra limitante poter definire funzioni aventi una singola espressione, in realtà si fa notare che in questo modo si è differenziato in modo netto lo scopo dei due oggetti (procedure e funzioni). Mentre le procedure servono per raccogliere istruzioni con compiti in comune, le funzioni servono come contenitore per espressioni complesse.

Anche in questo caso uno degli obiettivi è quello di rendere più leggibile il codice, nascondendo parte dell'elaborazione.

Anche le funzioni possono essere usate per la validazione, seppur generalmente ritornando un booleano come risultato e non un DataSet.

Esempi

```
create function M_GROUP_TYPE(GRP_TYP) {
    returns(
        if GRP_TYP = "1"
          then "X"
        elseif GRP_TYP = "2"
          then "B"
        elseif GRP_TYP = "3"
          then "I"
        elseif GRP_TYP = "4"
          then "O"
        else null
    ) as string
}
```

Questa funzione, presa direttamente da BIRD, non fa altro che mappare i valori del dominio di GRP_TYP nei valori di un altro dominio (CL_SHS_GROUP_TYPE).

Chiamata procedure/funzioni

Dato che questi due oggetti vengono usati in contesti differenti, anche il meccanismo con cui vengono invocati cambia leggermente.

Per le procedure, esiste la parola chiave **call**, a cui deve seguire il nome della procedura e, tra parentesi, le variabili che devono essere passate.

Sintassi

call *procname***(***varname [, varname]+***)**

Ovviamente, dobbiamo assicurarci che le variabili sia congruenti con i rispettivi parametri (l'ordine conta!). Inoltre dobbiamo ricordarci che l'ultima variabile è quella di output. Questo significa che l'ultima variabile verrà sovrascritta se esiste, creata altrimenti.

Come accennato nel capitolo relativo alla definizione di procedura, le istruzioni all'interno del corpo di una procedura sono conoscenza del contesto e quindi agiscono sulle variabili di conseguenza.

Facciamo quindi il caso delle clausole, e nello specifico la **keep**. Questa clausola prende in input i nomi delle colonne da mantenere. Come da specifiche VTL questi nomi non vengono passati come stringhe (come sarebbe ovvio), bensì come nome di variabile (più banalmente, senza virgolette).

Questo crea un problema: dato che il nome di colonna così fornito viene considerato come variabile, l'interprete cercherà quella variabile in memoria, fallendo, ovviamente. Per evitare la conseguente eccezione, la libreria evita la ricerca in memoria non procedendo nell'analisi ma fermandosi al nome. Leggendo il nome della variabile e convertendo quest'ultima in una stringa, si evita il problema e si può eseguire la clausola.

Arriviamo dunque alle procedure: quando definiamo una procedura, dobbiamo indicare il tipo di ogni parametro. Nel caso in cui volessimo indicare un nome di una colonna, siamo costretti ad usare il tipo *string*, anche se non è una variabile e certamente non è una variabile che contiene uno scalare di tipo stringa.

Il tipo di dato *string* usato nella definizione di variabile è quindi multiuso e può essere usato in diversi modi.

Lo stesso identico problema si presenta con la chiamata di funzione, ma con un livello di complessità minore, in quanto nella definizione di funzione non dobbiamo specificare il tipo di dato del parametro.

Dato che ne abbiamo parlato, vediamo quindi la sintassi della chiamata a funzione.

Sintassi
varname **:=** *funName***(***varname [, varname]**)**

Palesemente una funzione fa parte del gruppo delle espressioni, e quindi invece di assegnare il valore di ritorno ad una variabile, la stessa può essere usata in una espressione più complessa.

Se la cosa può sembrare un controsenso, dato che una funzione è utile proprio a nascondere i dettagli di un'espressione complessa, si fa notare che questa può beneficiare dall'uso delle funzioni, riducendo la complessità della stessa e aumentando la sua leggibilità.

A differenza delle procedure, dove bisogna verificare espressamente il tipo delle variabili passate come argomento, nelle funzioni il problema non si pone in questa fase (seppur rilevante durante la sua esecuzione). Supponendo che la funzione sia andata a buon fine, bisogna solo assicurarsi di usare il valore di ritorno nel modo più opportuno.

Ad es. se si usa una funzione che ritorna una stringa, non usare quel valore in una somma, pena interruzione dell'esecuzione