

SPARCL Tutorial, part 4

(Copyright 2002 by Lindsey Spratt. All Rights Reserved.)

3.1.2. Create the ‘Column Sum Query’/1 predicate. The section describes the creation of the clause to be used to query the ‘Column Sum’/3 clause built in the previous section. This is step 4 of the “Column Sum” example.

Step 4: Create the "Column Sum Query" clause.

The first two sub-steps of step 4 create the clause for the ‘Column Sum Query’/1 predicate with an empty body.

Step 4.1.

DO: Create a new clause named "Column Sum Query" with 1 argument in program (window) "Column Sum".

BY: Select the "Create Clause" option in the popup menu for the program. Remove 2 arguments from the default 3 arguments. Edit the default name ("Column Sum") to change it to "Column Sum Query".

Step 4.2: Create the argument term for the "Column Sum Query" clause.

The term in the argument of this query clause gives shape to the result of the query. The query clause that we will create provides a table of data and the name of a column to be summed in that table. The result of the query is a table of two rows: one row shows the test data and the other row shows the name of the column being summed and the sum for that column. The next step creates an empty table which is a set of one 2-tuple row. The interaction and result are shown in Figure 1 and Figure 2.

Step 4.3.

DO: Create a new term table in the argument of the “Column Sum Query” clause.

BY: Select the "Insert:Table" option in the popup menu for the argument.

The first column of this table is used to label the parts of the answer. The first row will hold the input “data”. Step 4.4 creates an ur constant “data” in the first column of the first row. Step 4.5 creates a variable which will corefer with the input table that will be placed in the first argument of the literal to be created in the query clause.

Step 4.4: Create a new ur constant of value “data” in the left term_table_cell.

Step 4.5.

DO: Create a new variable in the left term_table_cell.

BY: Select the "Insert:Variable" option in the popup menu for the left term_table_cell.

Step 4.6 creates a second row in the result table. This row will hold the sum of the selected column of the input data. The interaction and result of this step are shown in Figure 3 and

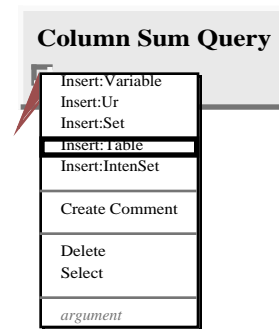


Figure 1: Interaction to create a table representation of a set.

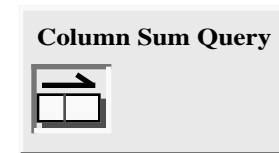


Figure 2: Result of creating a new table.

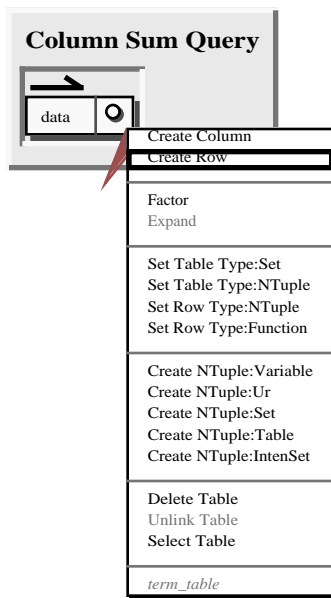


Figure 3: Interaction to add a second row to a term table.

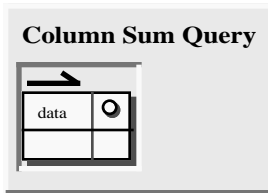


Figure 4: Result of adding a second row to a term table.

Figure 4.

Step 4.6.
 DO: Create a row at the bottom of the term_table.
 BY: Select the "Create Row" option in the popup menu for the term_table.

We want to ensure that the “data” row is displayed before (above) the “total” row, so we will make the result table an ordered table—one which is an N-tuple of rows instead of a set of rows. The interaction and result of this step are shown in

Figure 5 and Figure 6.

Step 4.7.
 DO: Set the "table type" of the term_table to ntuple.
 BY: Select the "Set Table Type:NTuple" option in the popup menu for the term_table.

Now we put the “label” for the second row in place. The interaction for starting this is shown in Figure 7.

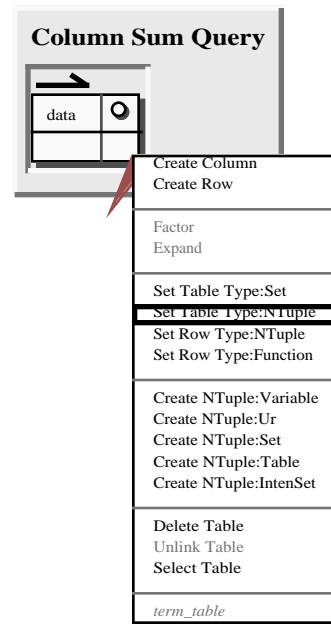


Figure 5: Interaction to set the type of a term table to “NTuple”.

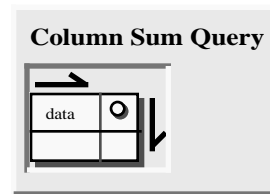


Figure 6: Result of setting the type of a term table to “NTuple”.

The resulting table is shown in Figure 8.

Step 4.8: Create a new ur constant of value “Calculation” in the term_table_cell.

Create a function term table.

The calculation from the ‘Column Sum’/3 predicate will be shown using a function term table with the column titles from the input data. It will have one row with the first column being the ur constant “Total” and the second column being the calculated total. The first step in constructing this new term table

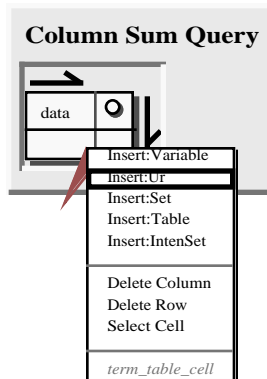


Figure 7: Interaction to insert an ur constant in a term table cell.

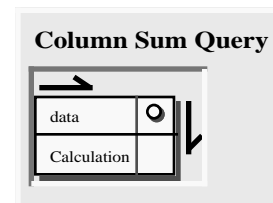


Figure 8: Result of inserting “Calculation” in a term table cell.

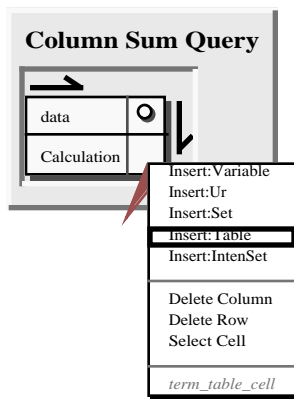


Figure 9: Interaction to create term table in a term table cell.

Step 4.9.
 DO: Create a new term table (call it term table 2) in the term_table_cell of row 2, column 2 of the table.
 BY: Select the "Insert:Table" option in the popup menu for this term_table_cell.

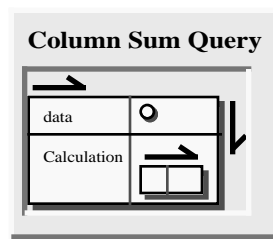


Figure 10: Result of creating a term table in a term table cell.

is shown in Figure 9 and Figure 10.

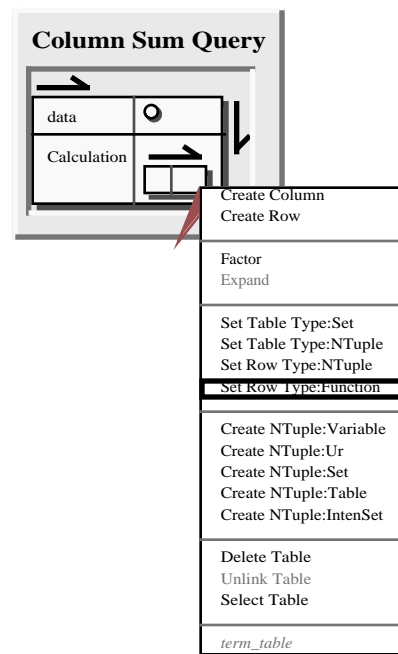


Figure 11: Interaction to set the row type of a term table to "Function".

The next step change the row type of the new term table to "Function", as shown in Figure 11 and Figure 12.

Step 4.10.
 DO: Set the "row type" of term_table 2 to function.
 BY: Select the "Set Row Type:Function" option in the popup menu for this term_table.

Step 4.11 creates a variable in the first column of the function table, as shown in Figure 13.

Step 4.11.
 DO: Create a new variable in the term_table_cell (1,1) of table 2.
 BY: Select the "Insert:Variable" option in the popup menu for this term_table_cell.

Steps 4.12, 4.13, and 4.14 create two more variables and the ur constant "Total" in the function table, as shown in Figure 14.

Step 4.12.
 DO: Create a new variable in the term_table_cell (1,2) of table 2.
 BY: Select the "Insert:Variable" option in the popup menu for this term_table_cell.

Step 4.13: Create a new ur constant of value "Total" in the term_table_cell (2,1) of table 2.

Step 4.14.
 DO: Create a new variable in the term_table_cell (2,2) of table 2.
 BY: Select the "Insert:Variable" option in the popup menu for this term_table_cell.

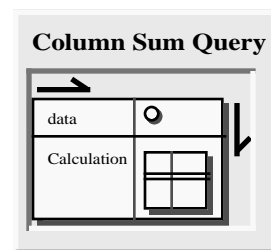


Figure 12: Result of setting the row type of a term table to "Function".

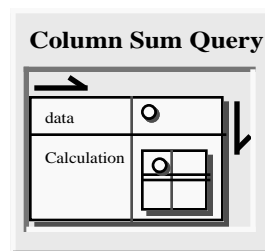


Figure 13: Result of creating a variable in the first column of the header of the function table.

Create a Column Sum literal.

The remaining major part of creating the clause defining the 'Column Sum Query'/1 predicate is creating the 'Column Sum'/3 literal with appropriately filled in arguments.

Step 4.15: Create the literal of the "Column Sum Query" clause.

The first sub-step is to create the literal with empty arguments.

Step 4.15.1.

DO: Create a new literal with name Column Sum and 3 arguments in the clause.

BY: Select the "Create Literal" option in the popup menu for the clause.

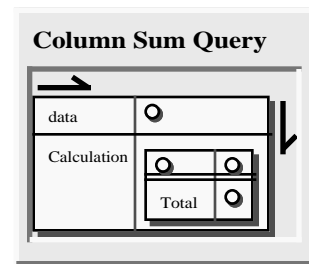


Figure 14: Complete result term in the argument of the clause defining the 'Column Sum Query'/1 predicate.

Having created the 'Column Sum'/3 literal, we now fill in its arguments. In the first argument we create a function table, put domain values in the "header" (the first row) of the table and fill in two "body" rows of data. First we'll create the empty function table, then we'll explain what a function table represents in SPARCL.

Step 4.15.2 creates a term table, shown in Figure 15.

Step 4.15.2.

DO: Create a new term table (call it table 3) in the first argument of the literal.

BY: Select the "Insert:Table" option in the popup menu for the argument.

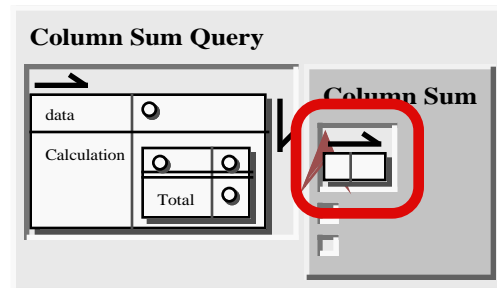


Figure 15: Result of creating a term table in first argument of literal.

Set up a function table in Column Sum.

The next step takes this ntuple-row-type table and converts it to a function-row-type table, shown in Figure 16.

Step 4.15.3.

DO: Set the "row type" of term_table 3 to function.

BY: Select the "Set Row Type:Function" option in the popup menu for this term_table.

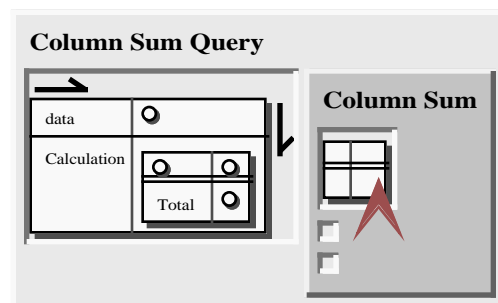


Figure 16: Result of setting the literal term table row type to "Function."

This conversion has added a row to the table so that it has a "header" row and one "body" row. The double horizontal line between the first and second row is the indication in the visual representation that this table is now a function table. It has the appearance of a table with named columns. A function table representation can be used for a set of "functions", where all of the functions in this set are finite and have the same domain values. These domain values are the

column names of the function table. The column names are placed at the top of the table, which we will do now in steps 4.15.4 and 4.15.5. The result is shown in Figure 17.

- Step 4.15.4: Create a new ur constant of value "Item" in term_table_cell (1,1) of table 3.
- Step 4.15.5: Create a new ur constant of value "Value" in term_table_cell (1,2) of table 3.

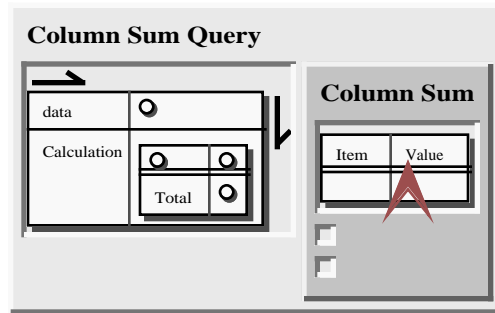


Figure 17: Result of adding the ur constants "Item" and "Value" as column headers to the function table in the literal.

Each "body" row plus the "header" row represents a function. A function in SPARCL is a set of ordered pairs (2-tuples), where the first elements of the ordered pairs are the domain values and the second elements are the range values. Further, no two ordered pairs in a "function" set have the same first element.

The range values of a function are placed here by steps 4.15.6 and 4.15.7. The result is shown in Figure 18.

- Step 4.15.6: Create a new ur constant of value "Sheet Music" in term_table_cell (2,1) of table 3.
- Step 4.15.7: Create a new ur constant of value "25.00" in term_table_cell (2,2) of table 3.

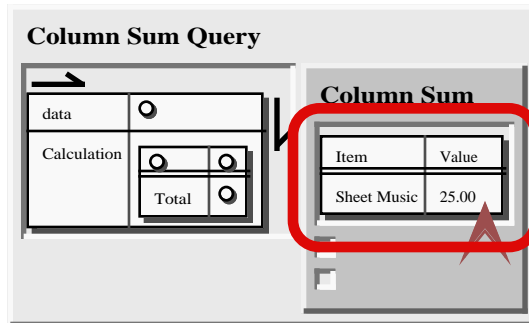


Figure 18: Result of adding the ur constants "Sheet Music" and "25.00" to the function table in the literal.

In step 4.15.8 we add another row to the table. "Adding" a row always puts it at the end of the table. For a table which is a set of rows (as opposed to an N-tuple of rows), this is semantically sufficient for available editing operations because the order of the rows doesn't make any semantic difference. But for a table which is an N-tuple of rows, this can be an awkward interface because there is no tool to place a new row at a particular place in the order of rows. To place a new row somewhere other than at the end, one would have to delete the rows after the place where the new row is wanted, add the new row, then rebuild the deleted rows.

- Step 4.15.8: DO: Create a row at the bottom of term_table 3. BY: Select the "Create Row" option in the popup menu for this term_table.

Next we fill in this last "body" row.

- Step 4.15.9: Create a new ur constant of value "Arm Chair & Stool" in term_table_cell (3,1) of table 3.
- Step 4.15.10: Create a new ur constant of value "75.00" in term_table_cell (3,2) of table 3.

Connect terms.

This completes the test data table. Now we put variables in the other two arguments. The result of this (overlain by a linking interaction) is shown in Figure 19.

- Step 4.15.11.
DO: Create a new variable in the second argument of the literal.
BY: Select the "Insert:Variable" option in the popup menu for this argument.
- Step 4.15.12.
DO: Create a new variable in the third argument of the literal.
BY: Select the "Insert:Variable" option in the popup menu for this argument.

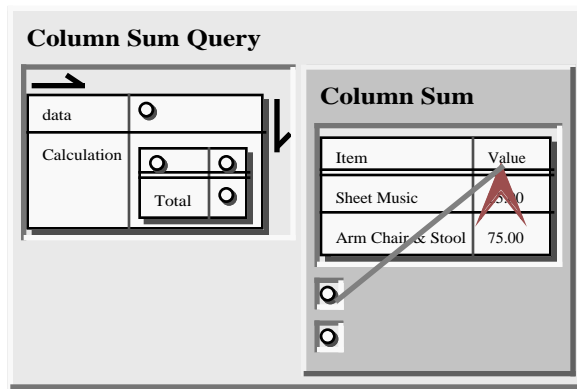


Figure 19: Interaction for linking "Value" to the argument for the name of the column to be summed.

Rather than put the same constant in the literal twice, we write it once and put a variable at the other location. Then we connect these two terms. In step 4.15.13, the title of the second column of the test data is connected to the variable in the literal argument for the name of the column to be summed. The interaction is shown in Figure 19. The result (overlain by the next interaction) is shown in Figure 20.

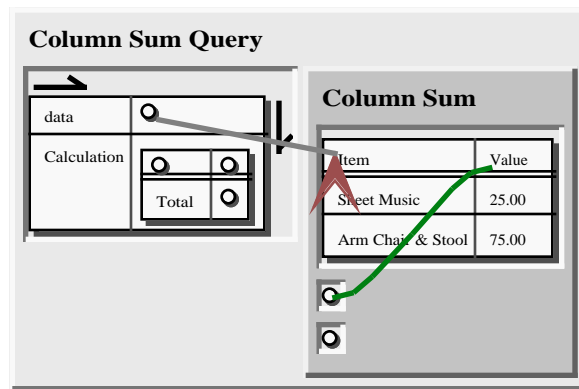


Figure 20: Result of linking "Value" to the argument for the name of the column to be summed, and interaction for linking the test data table to the test data cell variable of the result table.

- Step 4.15.13.
DO: Create a coreference link including the variable in the second argument of the literal and "Value".
BY: With "connect tool" as the current tool, depress the mouse button while the cursor is over the variable and drag the cursor until it's over "Value", then release the mouse button.

The entire table of test data is linked to the variable in the second column of the first row. The interaction is shown in Figure 20 and the result is shown in Figure 21.

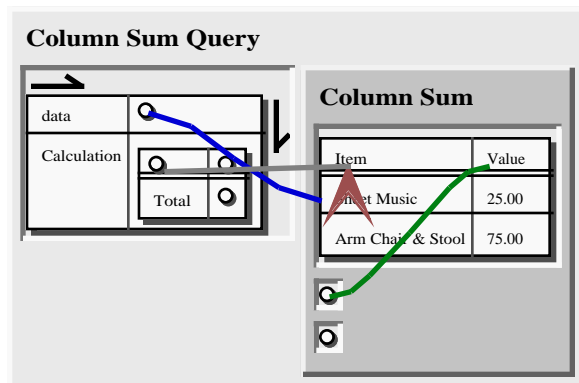


Figure 21: Result of linking the test data table to the test data cell variable of the result table, and interaction for linking "Item" to the head of the result data column for the "Total" label.

- Step 4.16.
DO: Create a coreference link including the variable in cell (1,2) of table 1 and all of table 2.
BY: With "connect tool" as the current tool, depress the mouse button while the cursor is over the variable and drag the cursor until it's over the table, then release the mouse button.

The name of the column not being

summed, “Item”, is linked to the head of the appropriate column in the result table’s function table. The interaction is shown in Figure 21 and the result is shown in Figure 22.

Step 4.17.

DO: Create a coreference link including the variable in cell (1,1) of table 2 and “Item”.

BY: With "connect tool" as the current tool, depress the mouse button while the cursor is over the variable and drag the cursor until it's over “Item”, then release the mouse button.

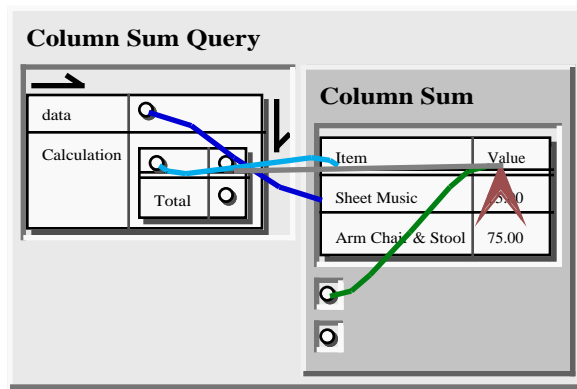


Figure 22: Result of linking “Item” to the to the head of the result data column *not* being summed, and interaction for linking “Value” to the result data column for the sum.

The name of the column being summed, “Value”, is linked to the head of the appropriate column in the result table’s function table. The interaction is shown in Figure 22 and the result is shown in Figure 23.

Step 4.18.

DO: Create a coreference link including the variable in cell (1,2) of table 2 and “Value”.

BY: With "connect tool" as the current tool, depress the mouse button while the cursor is over the variable and drag the cursor until it's over “Value”, then release the mouse button.

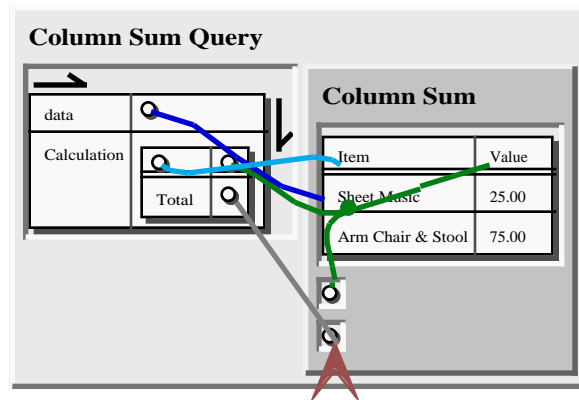


Figure 23: Result of linking “Value” to the result data column for the sum, and interaction for linking the column sum result variable to the result data column sum result value variable.

The final connection is from the result variable for the ‘Column Sum’/3 literal and the result sum variable in the second column of the “Total” row of the function table of the result table. The interaction is shown in Figure 23 and the result (plus the next interaction) is shown in Figure 24.

Step 4.19.

DO: Create a coreference link including the variable in cell (2,2) of table 2 and the variable in the third argument of the literal.

BY: With "connect tool" as the current tool, depress the mouse button while the cursor is over one variable and drag the cursor until it's over the other variable, then release the mouse button.

Finally, we will run the test query. The query interaction is shown in Figure 24. The result of the query is a 2-tuple with the name of the query clause as the first element and the bound value of the argument to the query clause as the second element. The query interaction is shown in Figure 24 and the result is in Figure 25.

Step 3.5.

DO: Execute a query of the "Column Sum Query" clause, with tracing information suppressed.
 BY: Select the "Query: Brief" option in the popup menu for the clause.

This query result in Figure 25 shows the sum of the items in the test data is 100.

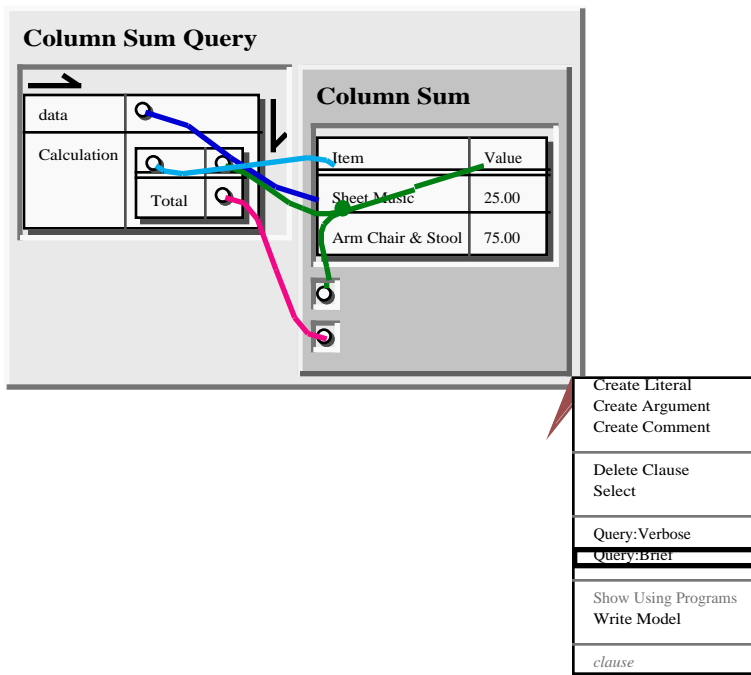


Figure 24: Interaction to query a 'Column Sum Query' clause.

Exercise

1. Exercise for section 3

1. Implement a "Tournament Scores" predicate which scores players based on their performance in multiple rounds of a tournament. In preparation for implementing this predicate (in part 3, below), you are first asked to implement "Maximal Range Value" in part 1, and then you are asked to implement

Item	Value
Arm Chair & Stool	75.00
Sheet Music	25.00

Item	Value
Total	100

Figure 25: Result of the query of the clause for 'Column Sum Query'.

"Maximal Pairs" in part 2. These predicates will make use of ordered pairs (2-tuples), partitioned sets, intensional sets (and intensional multisets), function tables, "*DELAY*" specifications, and the fails/2 metapredicate (a metapredicate is a predicate which takes a literal as an argument and invokes the given literal).

Write a "Maximal Range Value" predicate of two arguments.

1. Write a program for the predicate "Maximal Range Value" with two arguments. The first argument is a set of pairs, with the second elements (the range values) being numbers. The second argument is a number which is maximal with respect to the range values of this set, i.e. a number such that no range value is greater than it.

1. HINT:

Use a `fails/1` literal and use two literals in its argument, a `unify/2` literal and a `less/2` literal. The `unify/2` literal unifies given the pair set with a partitioned set with an ordered pair of two variables in one part and nothing in the other part. The `less/2` literal compares the second argument term with the second element of the ordered pair of the `unify/2` set.

2. Write a "Maximal Range Value Query" predicate of no arguments which tests the "Maximal Range Value/2" predicate by giving it the set `{a=>3,b=>4,c=>5}` and 5 as the maximal range value.

2. Write a "Maximal Pair" predicate of three arguments and a test predicate.

1. The "Maximal Pairs" predicate has a set of ordered pairs as its first argument, its second argument is the set of maximally range-valued pairs among the first argument's set. Thus, for the first argument set `{a=>2, b=>1, c=>2, d=>0}`, the second argument set of maximal pairs is `{a=>2, c=>2}`.

1. HINT:

This is implemented by a single clause, plus a "helper" predicate. The second argument to "Maximal Pairs" should be an intensional set. The template is an ordered pair (which will become the maximal ordered pairs). The body has two literals. One of these is a `unify/2` literal which unifies a 2-part partitioning with a variable. The first part contains a 2-tuple, the second part is hollow. The second literal is the "helper" predicate, "Maximal Range Value" of two arguments as implemented in part 1 of this exercise. For "Maximal Range Value"/2 to work correctly in "Maximal Pairs", there must be two `*DELAY*` specifications for the "Maximal Range Value" predicate of two arguments such that "Maximal Range Value" delays if either argument is a variable (i.e. a delay specification for `variable=>ignore` and another for `ignore=>variable`).

2. Write a "Maximal Pairs Query" predicate of one argument to test the "Maximal Pairs" predicate. The "Maximal Pairs Query" predicate should return the maximal pairs found by "Maximal Pairs", given the test set `{a=>2, b=>1, c=>2, d=>0}`.

3. Write a "Tournament Scores" predicate and a predicate to test it.

1. The "Tournament Scores" predicate has two arguments. The first argument is the tournament rounds scores and the second argument is a function from player to overall score for that player. The tournament rounds scores are a function table, where each column is a different player in the tournament and each row is a set of scores for a round of the tournament. The overall scoring of a player for the tournament is the number rounds in which that player was among those with the highest score.

1. HINT:

The second argument of "Tournament Scores" is an intensional multiset, where the template is a variable (which will be the player) and the body contains two literals. One of these literals is a unify/2 literal which extracts a row from the rounds table. The other literal is a "Maximal Pairs"/2 literal which finds the players with maximal scores for that round. The second argument to this "Maximal Pairs" literal should be a partitioned set of two parts. One of these parts is an ordered pair with the first element of this pair being connected to the template variable.

2. Write a "Tournament Scores Query" predicate of one argument which returns the overall player scores from "Tournament Scores", given a table of:

"{{a=>1, b=>2, c=>0}, {a=>2, b=>2, c=>1}, {a=>1, b=>0, c=>2}}".