OpenFile

Compound can write text files on disk. In order to write to a file a filepointer must be created. For this in *Compound* exists the command *OpenFile*.

Syntax:
filePtr = *OpenFile*(*Filename*, "*open mode*");

filePtr is a variable previously declared.

Filename can be a string or a variable of string type and represents the name of the file on disk.

There are two available *opening modes*:

- 'w'. In this mode a new file will be created and the user can write on it. If an old file with the same name exists, it's contents will be deleted.
- 'a'. In this mode if a file already exists, the user will append to what already exists in the file.

Example:

```
%Compound

# -------------------------------------------------------------
#      This is to check all available open and close
#                file options
# -------------------------------------------------------------
Variable myFilename    = "myFile.txt";
Variable file;

# ---------------------------
# First open for writing
# ---------------------------
file = openFile(myFilename, "w");
write2File(file, "This is the first time we write.\n");
closeFile(file);

# ---------------------------
# Re-open to append
# ---------------------------
file = openFile(myFilename, "a");
write2File(file, "This is the second time we write.\n");
closeFile(file);

End
```

Remove_Atom

Remove_atom removes an atom from a geometry given its index. We should point out that counting of atoms in ORCA starts with 0. After this command is executed it will store on a disk a new geomety in a xyz format where only the atom with the given index will be missing.

Syntax:

NewGeom = (Remove_atom, atomIndex, "filename", stepIndex, [geometry Index]);

where:
atom Index is the number of the atom we want to remove. It can be an integer number or a variable.
filename is the name of the file that we want to use for the new xyz file. It can be a string in quotation marks or a variable already defined before. In the name the xyz extension will be automatically appended. *step Index* the number of the step from which we will get the initial geometry. It has to be an integer number. *geometry Index* In case there are more than one geometries in the corresponding property file we can choose one. If no number is given but default the program will use the last one.

example:

if we use the normal ORCA input file:

```
*xyz 0 1
O         2.220871067         0.026716792         0.000620476
H         2.597492682        -0.411663274         0.766744858
H         2.593135384        -0.449496183        -0.744782026
*

%Compound "removeAtom.cmp"
```

together with the compound file "removeAtom.cmp" :

```
Variable filename  = "newGeom";
Variable atomIndex = 0;

New_Step
  !BP86
Step_End

New_Geom = ( Remove_atom, atomIndex, filename, 1);

end
```

then the xyz file 'newGeom.xyz' will be created that should look like:

```
2

   H         2.5974927       -0.4116633        0.7667449
   H         2.5931354       -0.4494962       -0.7447820
```

where the atom with *atomIndex = 0* meaning the first atom, meaning the oxygen is removed.

Remove_Element

Remove_element is similar to the Remove_atom but instead of using the index of the atom we use its atomic number. Thus the syntax is:

Syntax: NewGeom = (Remove_Element, atomic number, "filename", stepIndex, [geometry Index]);

where:
atomic number is the atomic number of the atom we want to remove. It can be an integer number or a variable.
filename is the name of the file that we want to use for the new xyz file. It can be a string in quotation marks or a variable already defined before. In the name the xyz extension will be automatically appended. *step Index* the number of the step from which we will get the initial geometry. It has to be an integer number. *geometry Index* In case there are more than one geometries in the corresponding property file we can choose one. If no number is given but default the program will use the last one.

example:

if we use again the input from paragraph *1.1.7.2* but instead of asking the compound file "removeAtom.cmp" we ask for the compound file "removeElement.cmp" that looks like:

```
Variable filename  = "newGeom";

New_Step
  !BP86
Step_End

New_Geom = ( Remove_element, 8, filename, 1);

end
```

then we will get again the same xyz file that was crated in paragraph *Remove_Atom* since the atom with atomic number 8 (meaning the Oxygen) will be removed from the original geometry.

NewStep

NewStep signals the beginning of a new ORCA input.

Syntax:

NewStep
…*Normal ORCA input commands*
StepEnd

There is no restriction in the input of ORCA, except of course that it should not include another Compound block. It is important to remember that a *NewStep* command should always end with a *Step_End* command. Below we show a simple example.

NOTE: The old syntax (New_Step/Step_End) is still compatible but please do not use it because in the future it will be deprecated.

Example:

```
NewStep
! BP86 def2-SVP
StepEnd
```

There is only a basic fundamental difference with a normal ORCA input. Inside the *NewStep* block it is not necessary to include a geometry. ORCA will automatically try to read the geometry from the previous calculation. Of course a geometry can be given and then ORCA will use it.

Print

Printing in the ORCA outpug can be customized using the *print* command. The syntax of the *print* command closely follows the corresponding printf command from C/C++. So the usage of the *print* command is:

Syntax:
print(format string, [variables]);

For each variable there can be specifiers and flags for the specifiers. Currently *print* command supports three datatypes namely integers, doubles and strings.

A format specifier follows this prototype: *%[flags][width][.precision]specifier*

where details for the specifiers and flags can be found in table Table 8.1

Table 8.1: compound print Specifiers

Specifier	
s	strings
d	integers
lf	doubles
Flags	
number	width
.number	number of decimal digits
-	left alignement (by default is right)

Example:

```
# -----------------------------------------------------------
# This is to check all available print defintions
# -----------------------------------------------------------
Variable x1    = 2.0;
Variable x2    = {10.0, 20.0, 30.0, 40.0};
Variable x3    = {10,   20,   30,   40};
Variable x4    = {"ten", "twenty", "thirty", "fourty"};
Variable x5 = "test";
Variable index = 2;

print( " --------------------------------------------- \n");
print( " ------ SUMMARY OF PRINT DEFINITIONS --------- \n");
print( " --------------------------------------------- \n");
# ---------- No variables ------------
print( " No variables: \n" );
# ------------- Doubles --------------
print( " -------- Doubles --------------\n");
print( " constant double ( no format)             : %lf\n", 3.5);
print( " constant double (defined width)          : %16lf\n",3.5);
print( " constant double (defined width/accuracy) : %16.8lf\n", 3.5);
print( " variable double (x1)                     : %lf\n", x1);
print( " function double 2*x1*x1                  : %lf\n", 2*x1*x1);
print( " array element double                     : %lf\n", x2[2]);
print( " array element double with var index      : %lf\n", x2[index]);
#print( " array element double with function index : %lf\n", x2[index + 1];
# ------------ Integers --------------
print( " -------- Integers --------------\n");
print( " constant integer ( no format)            : %d\n", 3);
print( " constant integeer (defined width)        : %8d\n",3);
print( " variable integer (index)                 : %d\n", index);
print( " function integer 2*index*index           : %d\n", 2*index*index);
print( " array element                            : %d\n", x3[2]);
print( " array element intteger with var index    : %lf\n",x3[index]);
# ------------  Strings --------------
```

(continues on next page)

(continued from previous page)

```
print( " -------- Strings --------------\n");
print( " constant string ( no format)           : %s\n", "test");
print( " constant string (defined width)        : %8s\n","test");
print( " variable string                        : %s\n", x5);
print( " array element                          : %s\n", x4[2]);
print( " array element intteger with var index  : %s\n",x4[index]);
print(" ----------------------------------------------\n");
print(" ----------------------------------------------\n");
End
```

ReadProperty

One of the fundamental features of *Compound* is the ability to easily read ORCA calculated values from the property file.

Syntax:

[res=] readProperty(propertyName=myName, [stepID=myStep], [filename=myFilename], [baseProperty=true/false])

Where: res: An integer that returns the index of the found property if the property was found in the property file, -1 if the property does not exist. This is not obligatory.

propertyName: A string alias that defines the variable the user wants to read.

stepID: The step from which we want to read the property. If not given the property file from the last step will be read.

filename: A filename of a property file. If a filename and at the same time a step are provided the program will ignore the step and try to read the property file with the given filename.

NOTE please note that in the end of filename the extension *.property.txt* will be added.

baseProperty: A true/false boolean. The default value is set to false. If the value is set to true then a generic property of the type asked will be read. This means if dipole moment is asked, it will return the last dipole moment, irrelevant if and MP2 or SCF one wad defined.

Example

```
# ----------------------------------------------------
# This is an example script for readProperty
# -----------------------------------------------------
%Compound
  Variable enDirect=0.0;
  Variable enFilename=0.0;
  Variable myProperty="DFT_Total_en";
  Variable basename="compound_example_propererttFile_readProperty";
  Variable newBasename ="newFilename";
  Variable res = -1;
  NewStep
    !BP86
    *xyz 0 1
      H 0.0 0.0 0.0
      H 0.0 0.0 0.8
    *
  StepEnd
  NewStep
    !B3LYP
  StepEnd
  # First read the energy directly
  res = enDirect.ReadProperty(propertyName=myProperty, stepID=2);
  print("res : %d\n", res);
```

(continues on next page)

(continued from previous page)

```
  # Now read the same energy through filename
  SysCmd("cp %s_Compound_2.property.txt %s.property.txt", basename, newBasename);
  enFilename.ReadProperty(propertyName=myProperty, filename=newBasename);
  print(" Energy direct    : %.9lf\n", enDirect);
  print(" Energy from file : %.9lf\n", enFilename);
  print("Difference between 2 energies : %.12lf\n", enDirect-enFilename);
End
```

Read_Geom

Read_Geom will read the geometry from a previous step.

Syntax:
Read_Geom *number*

Here number is the number of the job that we want to read the geometry from. The directive should be positioned before a *NewStep* - *StepEnd* block.

Example:

```
#Compound Job 1
New_Step
!BP86 def2-SVP
Step_End

#Compound Job 2
New_Step
!BP86 def2-SVP opt
Step_End

#Compound Job 3
Read_Geom 1
New_Step
!CCSD  def2-SVP
Step_End

End  #Final End
```

In this case the third calculation, through the *Read_MGeom 1* command, will read the geometry from the first calculation.

ReadMOs

ReadMOs reads the molecular orbitals from a previous step.

Syntax:
ReadMOs(stepNumber);

Where:

- *stepNumber:* is the number of the step from which we want to read the orbitals.

Example:

```
# ---------------------------------------------------
# This is an example script for ReadMOs
# ----------------------------------------------------
%Compound
  Variable step = 1;
  New_Step
```

(continues on next page)

(continued from previous page)

```
    !BP86
    *xyzfile 0 1 h2o.xyz
  Step_End

  ReadMOs(step);
  New_Step
    !BP86
  Step_End
End
```

S.GetBasename

In *Compound* strings have all the functionality of a normal variable. In addition they have some additional functions that act only on strings. On of these functions is the function *GetBasename*. This function searches the string and if it contains a dot it will return the part of the string before the dot.

Syntax:

result = source.GetBasename();

Where:

result is the returned string.

source is the original string.

NOTE If the original string contains no dot then the result string will be a copy of the source one.

Example:

```
# ------------------------------------------------------
# This is an example script for string related functions:
#     - GetBasename
#     - GetSuffix
#     - GetChar
# ------------------------------------------------------
%Compound
  Variable original = "lala.xyz";
  Variable basename, suffix;
  Variable constructed = "";
  basename = original.GetBasename();
  suffix   = original.GetSuffix();
  for i from 0 to original.stringlength()-1 Do
    write2String(constructed,"%s%s", constructed, original.GetChar(i));
  endfor
  print("Original    : %s\n", original);
  print("Basename    : %s\n", basename);
  print("Sufix       : %s\n", suffix);
  print("Constructed : %s\n", constructed);
End
```

S.GetChar

In *Compound* strings have all the functionality of a normal variable. In addition they have some additional functions that act only on strings. On of these functions is the function *GetChar*. This function searches the string and if it contains a dot it will return the part of the string after the dot.

Syntax:

result = source.GetChar(index);

Where:

result is the returned string.

index is the index of the curracter in the string. Keep in mind that counting starts with **0** and not 1.

source is the original string.

NOTE If the index is larger than the size of the string or negative then the program will exit.

Example:

```
# -------------------------------------------------------
# This is an example script for string related functions:
#     - GetBasename
#     - GetSuffix
#     - GetChar
# -------------------------------------------------------
%Compound
  Variable original = "lala.xyz";
  Variable basename, suffix;
  Variable constructed = "";
  basename = original.GetBasename();
  suffix   = original.GetSuffix();
  for i from 0 to original.stringlength()-1 Do
    write2String(constructed,"%s%s", constructed, original.GetChar(i));
  endfor
  print("Original    : %s\n", original);
  print("Basename    : %s\n", basename);
  print("Sufix       : %s\n", suffix);
  print("Constructed : %s\n", constructed);
End
```

S.GetSuffix

In *Compound* strings have all the functionality of a normal variable. In addition they have some additional functions that act only on strings. On of these functions is the function *GetSuffix*. This function searches the string and if it contains a dot it will return the part of the string after the dot.

Syntax:

result = source.GetSuffix();

Where:

result is the returned string.

source is the original string.

NOTE If the original string contains no dot then the result string will be an empty string.

Example:

```
# -------------------------------------------------------
# This is an example script for string related functions:
#     - GetBasename
```

(continues on next page)

(continued from previous page)

```
#     - GetSuffix
#     - GetChar
# -----------------------------------------------------
%Compound
  Variable original = "lala.xyz";
  Variable basename, suffix;
  Variable constructed = "";
  basename = original.GetBasename();
  suffix   = original.GetSuffix();
  for i from 0 to original.stringlength()-1 Do
    write2String(constructed,"%s%s", constructed, original.GetChar(i));
  endfor
  print("Original    : %s\n", original);
  print("Basename    : %s\n", basename);
  print("Sufix       : %s\n", suffix);
  print("Constructed : %s\n", constructed);
End
```

StepEnd

StepEnd signals the end of an ORCA Input. It should always be the last directive of an ORCA input inside the compound block that starts with *New_Step* (see paragraph *NewStep*)

NOTE: The old syntax (New_Step/Step_End) is still compatible but please do not use it because in the future it will be deprecated.

Sys_cmd

Sys_cmd will read a system command and execute it.

Syntax:
Sys_cmd *command*

Example:

```
SYS_CMD "orca_mapspc test.out SOCABS -x0700 -x1900 -w0.5 -eV -n10000 "
```

Timer

A *timer* is an object that can keep time for tasks in compound. Before a timer object is used it has to be declared. The declaration of a *timer* is slightly different that the rest of variables, because it has to explicitly declare it's type.

Syntax

timer myTimer;

where

timer is used instead of the normal *variable* command to explicitly set the variable type to *compTimer*.

myTimer is a normal instance of the object.

Example:

```
# ----------------------------------------------
# This is to test timer functions.
# --------------------------------
timer tm;
Variable x = 0.0;
tm.start();
```

(continues on next page)

(continued from previous page)

```
for index from 0 to 100000 Do
  x = x + 0.1;
EndFor
tm.stop();
x = tm.Total();
print( "-------------------------------\n");
print( "   Compound - Timer Results   \n");
print( "-------------------------------\n");
print( " First total time:   %.2lf\n", x);

x = tm.total();
tm.Reset();
tm.Start();
for index from 0 to 200000 Do
  x = x + 0.1;
EndFor
tm.Stop();
x = tm.total();

print( " Second total time:   %.2lf\n", x);
End
```

Below is a lit of functions that work exclusively on *Timer* objects.

- Last (*T.Last*)
- Reset (*T.Reset*)
- Start (*T.Start*)
- Stop (*T.Stop*)
- Total (*T.Total*)

T.Last

Last is a function that works on a *timer* object. It returns, as a real number, the last value of the timer.

Syntax

myTimer.Last();

Where:

myTimer: is the *timer* object initialized before.

Example:

```
# -----------------------------------------------
# This is to test timer functions.
# --------------------------------
timer tm;
Variable x = 0.0;
tm.start();
for index from 0 to 100000 Do
  x = x + 0.1;
EndFor
tm.stop();
x = tm.Total();
print( "-------------------------------\n");
print( "   Compound - Timer Results   \n");
print( "-------------------------------\n");
print( " First total time:   %.2lf\n", x);
```

(continues on next page)

(continued from previous page)

```
x = tm.total();
tm.Reset();
tm.Start();
for index from 0 to 200000 Do
  x = x + 0.1;
EndFor
tm.Stop();
x = tm.total();

print( " Second total time:   %.2lf\n", x);
End
```

NOTE Before using last the *timer* object must, beside defined, be also initializee, using *Start* (see *T.Start*)

T.Reset

Reset is a function that works on a *timer* object. It resets the *timer* object to its initial state.

Syntax

myTimer.Reset();

Where:

myTimer: is the *timer* object initialized before.

Example:

```
# -----------------------------------------------
# This is to test timer functions.
# --------------------------------
timer tm;
Variable x = 0.0;
tm.start();
for index from 0 to 100000 Do
  x = x + 0.1;
EndFor
tm.stop();
x = tm.Total();
print( "------------------------------\n");
print( "   Compound - Timer Results   \n");
print( "------------------------------\n");
print( " First total time:   %.2lf\n", x);

x = tm.total();
tm.Reset();
tm.Start();
for index from 0 to 200000 Do
  x = x + 0.1;
EndFor
tm.Stop();
x = tm.total();

print( " Second total time:   %.2lf\n", x);
End
```

T.Start

Start is a function that works on a *timer* object. It retunrs the *timer* object to its initial state.

Syntax

myTimer.Start();

Where:

myTimer: is the *timer* object initialized before.

Example:

```
# -----------------------------------------------
# This is to test timer functions.
# --------------------------------
timer tm;
Variable x = 0.0;
tm.start();
for index from 0 to 100000 Do
  x = x + 0.1;
EndFor
tm.stop();
x = tm.Total();
print( "------------------------------\n");
print( "   Compound - Timer Results   \n");
print( "------------------------------\n");
print( " First total time:   %.2lf\n", x);

x = tm.total();
tm.Reset();
tm.Start();
for index from 0 to 200000 Do
  x = x + 0.1;
EndFor
tm.Stop();
x = tm.total();

print( " Second total time:   %.2lf\n", x);
End
```

T.Stop

Stop is a function that works on a *timer* object. It stops the timer from counting.

Syntax

myTimer.Stop();

Where:

myTimer: is the *timer* object initialized before.

Example:

```
# -----------------------------------------------
# This is to test timer functions.
# --------------------------------
timer tm;
Variable x = 0.0;
tm.start();
for index from 0 to 100000 Do
  x = x + 0.1;
```

(continues on next page)

(continued from previous page)

```
EndFor
tm.stop();
x = tm.Total();
print( "------------------------------\n");
print( "   Compound - Timer Results   \n");
print( "------------------------------\n");
print( " First total time:   %.2lf\n", x);

x = tm.total();
tm.Reset();
tm.Start();
for index from 0 to 200000 Do
  x = x + 0.1;
EndFor
tm.Stop();
x = tm.total();

print( " Second total time:   %.2lf\n", x);
End
```

T.Total

Total is a function that works on a *timer* object. It returns a real number with the total time.

Syntax

myTimer.Total();

Where:

myTimer: is the *timer* object initialized before.

Example:

```
# -----------------------------------------------
# This is to test timer functions.
# --------------------------------
timer tm;
Variable x = 0.0;
tm.start();
for index from 0 to 100000 Do
  x = x + 0.1;
EndFor
tm.stop();
x = tm.Total();
print( "------------------------------\n");
print( "   Compound - Timer Results   \n");
print( "------------------------------\n");
print( " First total time:   %.2lf\n", x);

x = tm.total();
tm.Reset();
tm.Start();
for index from 0 to 200000 Do
  x = x + 0.1;
EndFor
tm.Stop();
x = tm.total();

print( " Second total time:   %.2lf\n", x);
End
```

Variables - General

Everything in the *Compound* language is based on variables. Their meaning and usage are similar to those in any programming language: you need to declare a variable and then assign a value to it. Notably, in *Compound*, a variable must be declared before it is assigned a value, following the syntax rules of languages like C. This differs from languages like Python, where you can assign a value to a variable without prior declaration. The only exception to this rule in Compound is the index in a for loop, which does not require prior declaration.

In *Compound* we support the following data types for variables:

- Integer
- Double
- String
- Boolean
- File pointer

In addition to these data types *Compound* supports also variables of type *Geometry* and *Timer* but these are treated separately (see *Geometry* and *Timer*).

For each variable in *Compound* there are 3 major categories of usage:

- The declaration (see *Variables - General*)
- The assignement (see *Variables - Assignment*) and
- Variable functions (see *Variables - Functions*)

Variables - Declaration

There are currently 6 different ways to declare a variable in *Compound*. Their syntax is the following:

Syntax:

A. *Variable name;*

B. *Variable name1, name2;*

C. *Variable name=value;*

D. *Variable name[n];*

E. *Variable name[$n1$][$n2$];*

F. *Variable name={value1, value2, …}*;

NOTE In previous versions of *Compound* for variables that were matrices but the size was not known one had to declare the variable using the following syntax:

Variable name = [];

This is now changed and the empty brackets are no longer needed, so that this variable can be defined like a normal variable as in case *A*.

Example

```
# -------------------------------------------------------------
# This is to check all available ways of
#   Variable declaration in Compound
# -------------------------------------------------------------
Variable size = 5;
Variable x1;                              #caseA
Variable x2,x3;                           #caseB
Variable x4 = 1.0;                        #caseC
Variable x5 = 0;                          #caseC
```

(continues on next page)

(continued from previous page)

```
Variable x6 = "Test";                    #caseC
Variable x7 = True;                      #caseC
Variable x8 = 2*x4;                      #caseC
Variable x9 = x6;                        #caseC
Variable x10 = x7;                       #caseC
Variable x11;                            #caseA
Variable x12[3];                         #caseD
Variable x12b[size-2];                   #caseD
Variable x12c[size][size-1];             #caseE
x12b[1] = 4.0;
x12c[2][2] = 7.0;
Variable x13[3][3];                      #caseE
Variable x14 = {2.0, 4*x4, 2, "lala"};   #caseF
Variable x15 = {0, 1, 2, 3, 4};
Variable x15b = {0, 1, 2, 3, 4};
Variable x15c[x15[x15b[2]-1]+2];
Variable x16, x17=x10, x18;
Variable x19=2.0, x20, x21[2], x23[size][size];
print( " --------------------------------------------- \n");
print( " ---------- SUMMARY OF DEFINITIONS ----------- \n");
print( " --------------------------------------------- \n");
print( " x4 (1.0)          :  %.2lf\n", x4);
print( " x5 (0)            :  %.2d\n",  x5);
print( " x6 (\"Test\")       :  %s\n",    x6);
if (x7) then
  print(" x7 (True)         :  TRUE\n");
else
  print(" x7 (True)         :  FALSE\n");
endIf
print(" x8 (2*x4)         :  %.2lf\n", x8);
print(" x9 (x6)           :  %s\n", x9);
print(" x12b[1] (4.0)     :  %lf\n", x12b[1]);
print(" x12c[2][2] (7.0)  :  %lf\n", x12c[2][2]);
print(" Variable x14 = {2.0, 4*x4, 2, \"lala\"};\n");
print(" x14[0]            :  %lf\n", x14[0]);
print(" x14[1]            :  %lf\n", x14[1]);
print(" x14[2]            :  %d\n",  x14[2]);
print(" x14[3]            :  %s\n",  x14[3]);
print(" Variable x15  = {0, 1, 2, 3, 4};\n");
print(" Variable x15b = {0, 1, 2, 3, 4};\n");
print(" Variable x15c[x15[x15b[2]-1]+2];\n");
print(" x15c.GetSize()    :  %d\n",  x15c.GetSize());
print(" ---------------------------------------------\n");
print(" ---------------------------------------------\n");
End
```

Some comments for the different cases of variable declaration.
Case A is the simplest one were we just declare the name of a variable.

Case B is similar to *Case A* but here more than one variables are declared simultaneously.

In **Case C** we combine the variable declaration with the assignment of a value to the variable. It worth noting that in this case *Compound* automatically deducts the type of the variable based on the given value.

Case D declares a 1-Dimensional array of a defined size.

Case E declares a 2-Dimensional array of defined size. For this case, and also accordingly for *Case E*, one can use previously defined integer variables instead of numbers.

Case F defines an array based on a list of given values. The array will automatically define it's size based on the size of the list. The values in the list do not have to be all of the same type.

NOTE In the past for *Case F* empty brackets were needed after the name of the variable. This is no longer necessary.

NOTE It is important not to forget the final **;** symbol in the end of each declaration because the result of omitting it is undefined.

Variables - Assignment

Assigning a value to a variable has a rather straightforward syntax.

Syntax:
VariableName = CustomFunction;

Where:

VariableName is a variable already declared.

CustomFunction a mathematical expression.

Example:

```
# -------------------------------------------------------------
# This is to check all available ways of variable assignement
#   (It does not take care of 'with' we will have a separate
#         file for this)
# -------------------------------------------------------------

# -----------------------------------
# Some necessary initial declarations
# -----------------------------------
Variable x1, x2, x3, x4;
Variable y1, y2, y3, y4;
Variable x5[4];
Variable y5[4];
Variable x6[3][3];
Variable y6[3][3];

# -----------------------------------
# Now the assignements
# -----------------------------------
#Scalars doubles
x1 = 1.0;
y1 = 2*x1;
#Scalars integers
x2 = 1;
y2 = 2*x2;
#Scalars strings
x3 = "test";
y3 = x3;
#Scalars bools
x4 = True;
y4 = x4;
#1D Arrays
x5[0] = 2.0;
y5[0] = x5[0];
x5[1] = 1;
y5[1] = 2*x5[1];
x5[2] = "test";
y5[2] = x5[2];
x5[3] = True;
y5[3] = x5[3];
#2D Arrays
x6[0][0] = 1.0;
y6[0][0] = 2*x6[0][0];
print( " --------------------------------------------- \n");
print( " ---------- SUMMARY OF ASSIGNMENTS ---------- \n");
```

(continues on next page)

(continued from previous page)

```
print( " -------------------------------------------- \n");
print( " ---------------- Scalars -------------------- \n");
print( " x1         :  %.2lf\n", x1);
print( " y1         :  %.2lf\n", y1);
print( " x2         :  %d\n", x2);
print( " y2         :  %d\n", y2);
print( " x3         :  %s\n", x3);
print( " y3         :  %s\n", y3);
print( " ---------------- 1D - Arrays----------------- \n");
print( " x5[0]      :  %.2lf\n", x5[0]);
print( " y5[0]      :  %.2lf\n", y5[0]);
print( " x5[1]      :  %d\n", x5[1]);
print( " y5[1]      :  %d\n", y5[1]);
print( " x5[2]      :  %s\n", x5[2]);
print( " y5[2]      :  %s\n", y5[2]);
print( " ---------------- 2D - Arrays----------------- \n");
print( " x6[0][0]   :  %lf\n", x6[0][0]);
print( " y6[0][0]   :  %lf\n", y6[0][0]);
print(" ----------------------------------------------\n");
print(" ----------------------------------------------\n");
End
```

NOTE It is important to remember to finish the variable assignment using the **‘;’** symbol.

Variables - Functions

Variables in *Compound* have a small number of functions that can help exctract information about them. In the current version of *Compound* these functions are the following:

Syntax:
VariableName.Function();

where *VariableName* is a variable that is already declared. Then the function will return a value that depends on the *Function* that we used.

Currenlty *Compound* supports the following functions:

- GetBool() *V.GetBool()*
- GetDim1() *V.GetDim1()*
- GetDim2() *V.GetDim2()*
- GetDouble() *V.GetDouble()*
- GetInteger() *V.GetInteger()*
- GetSize() *V.GetSize()*
- GetString() *V.GetString()*
- PrintMatrix() *V.PrintMatrix()*

Example:

```
# -------------------------------------------------------------
# This is to check all available ways of variable assignement
#   (It does not take care of 'with' we will have a separate
#        file for this)
# -------------------------------------------------------------

# -----------------------------------
# Some necessary initial declarations
# -----------------------------------
```

(continues on next page)

(continued from previous page)

```
Variable x1, x2, x3, x4;
Variable y1, y2, y3, y4;
Variable x5[4];
Variable y5[4];
Variable x6[3][3];
Variable y6[3][3];

# -----------------------------------
# Now the assignements
# -----------------------------------
#Scalars doubles
x1 = 1.0;
y1 = 2*x1;
#Scalars integers
x2 = 1;
y2 = 2*x2;
#Scalars strings
x3 = "test";
y3 = x3;
#Scalars bools
x4 = True;
y4 = x4;
#1D Arrays
x5[0] = 2.0;
y5[0] = x5[0];
x5[1] = 1;
y5[1] = 2*x5[1];
x5[2] = "test";
y5[2] = x5[2];
x5[3] = True;
y5[3] = x5[3];
#2D Arrays
x6[0][0] = 1.0;
y6[0][0] = 2*x6[0][0];
print( " --------------------------------------------- \n");
print( " ---------- SUMMARY OF ASSIGNMENTS ---------- \n");
print( " --------------------------------------------- \n");
print( " ---------------- Scalars -------------------- \n");
print( " x1          :  %.2lf\n", x1);
print( " y1          :  %.2lf\n", y1);
print( " x2          :  %d\n", x2);
print( " y2          :  %d\n", y2);
print( " x3          :  %s\n", x3);
print( " y3          :  %s\n", y3);
print( " ---------------- 1D - Arrays---------------- \n");
print( " x5[0]       :  %.2lf\n", x5[0]);
print( " y5[0]       :  %.2lf\n", y5[0]);
print( " x5[1]       :  %d\n", x5[1]);
print( " y5[1]       :  %d\n", y5[1]);
print( " x5[2]       :  %s\n", x5[2]);
print( " y5[2]       :  %s\n", y5[2]);
print( " ---------------- 2D - Arrays---------------- \n");
print( " x6[0][0]    :  %lf\n", x6[0][0]);
print( " y6[0][0]    :  %lf\n", y6[0][0]);
print(" ----------------------------------------------\n");
print(" ----------------------------------------------\n");
End
```

V.GetBool()

This function will return a boolean value in case the variable is boolean or integer. For integers it will return *false* for 0 and *true* for all other integer values. In all other cases the program will crash providing a relevant message.

Syntax:

myVar.GetBool();

where:

myVar is an already initialized variable.

Example

```
# -----------------------------------------------------
# This is an example script for
#  Variable functions
# ------------------------------------------------------
%Compound
  Variable double=1.0;
  Variable integer=2;
  Variable iToBool = integer.GetBool();
  Variable boolean=false;

  print("----------------------------------------\n");
  print("     Results for translation functions \n");
  print("Double  to integer : %d    (it should print 1)\n", double.GetInteger());
  print("Integer to double  : %.2lf (it should print 2.00)\n", integer.
↪GetDouble());
  print("Boolean to string  : %s    (it should print FALSE)\n", boolean.
↪GetString());
  print("Integer to boolean : %s    (it should print TRUE)\n", iToBool.
↪GetString());
  print("Double  to string  : %s    (it should print 1.00000000000000000000e+00)\n
↪", double.GetString());
  print("Integer to string  : %s    (it should print 2)\n", integer.GetString());
End
```

V.GetDim1()

This function works on a variable. It will return the size of the first dimension of an array. If the variable is a scalar it will return 1.

Syntax:

myVar.GetDim1();

where:

myVar is an already initialized variable.

Example

```
# -----------------------------------------------------
# This is an example script for
#  Variable functions
# ------------------------------------------------------
%Compound
  Variable dim1, dim2, size;
  Variable A;
  Variable B[3];
  Variable C[3][2];
```

(continues on next page)

(continued from previous page)

```
  print("----------------------------------------\n");
  print("          Results for scalar  \n");
  print("Dim1 : %d (it should print 1)\n", A.GetDim1());
  print("Dim2 : %d (it should print 1)\n", A.GetDim2());
  print("Size : %d (it should print 1)\n", A.GetSize());
  print("----------------------------------------\n");
  print("          Results for 1D-Array  \n");
  print("Dim1 : %d (it should print 3)\n", B.GetDim1());
  print("Dim2 : %d (it should print 1)\n", B.GetDim2());
  print("Size : %d (it should print 3)\n", B.GetSize());
  print("----------------------------------------\n");
  print("          Results for 2D-Array  \n");
  print("Dim1 : %d (it should print 3)\n", C.GetDim1());
  print("Dim2 : %d (it should print 2)\n", C.GetDim2());
  print("Size : %d (it should print 6)\n", C.GetSize());
End
```

V.GetDim2()

This function works on a variable. It will return the size of the second dimension of an array. If the variable is a scalar or a 1-Dimensional array it will return 1.

Syntax:

myVar.GetDim2();

where:

myVar is an already initialized variable.

Example

```
# ----------------------------------------------------
# This is an example script for
#  Variable functions
# -----------------------------------------------------
%Compound
  Variable dim1, dim2, size;
  Variable A;
  Variable B[3];
  Variable C[3][2];

  print("----------------------------------------\n");
  print("          Results for scalar  \n");
  print("Dim1 : %d (it should print 1)\n", A.GetDim1());
  print("Dim2 : %d (it should print 1)\n", A.GetDim2());
  print("Size : %d (it should print 1)\n", A.GetSize());
  print("----------------------------------------\n");
  print("          Results for 1D-Array  \n");
  print("Dim1 : %d (it should print 3)\n", B.GetDim1());
  print("Dim2 : %d (it should print 1)\n", B.GetDim2());
  print("Size : %d (it should print 3)\n", B.GetSize());
  print("----------------------------------------\n");
  print("          Results for 2D-Array  \n");
  print("Dim1 : %d (it should print 3)\n", C.GetDim1());
  print("Dim2 : %d (it should print 2)\n", C.GetDim2());
  print("Size : %d (it should print 6)\n", C.GetSize());
End
```