An introduction to GCL

The OpenGenie Tutorials

Kenneth Shankland
March 1996

Objectives

The object of this tutorial is to introduce the user to most of the aspects of Genie Command Language (GCL) programming. It is GCL language that gives OpenGenie its great flexibility relative to Genie V2.

Requirements

These sheets and a copy of the file "gcl_tutorial_ks.gcl". The file contains all the procedures that you will use, so you don't have to type them in ! Where text that you have type is shown in upper case, you should adhere to this. I have put comments after some lines. The comments begin with a '#' character. You do not need to type them, though you could if you really wanted to, because Genie will ignore them anyway. Don't forget that you can use the up arrow to recall previous lines and edit them.

Getting started

Start up genie and load the gcl_tutorial_ks.gcl file:-

load "gcl_tutorial_ks.gcl"

If you (or Genie !) gets confused at any time, just exit from the program by typing 'exit' the start up again and reload the gcl tutorial file.

Variables and their types

At the interactive level, variables can be created freely. Their type is determined by the items assigned to them and the type can be changed at any time. Try the following:-

```
i = 3 ; printn i
i = 3.4 ; printn i
i = "vic" ; printn i
```

Note that you have typed two individual GCL commands on each single line, using the ';' character as a separator. The printn command prints variable i and then prints a newline character - this is distinct from the print command which does not move on to a newline (a bit like the $ sign in a Fortran format statement). The variable i is first an integer, then a real, then a string. There is no problem changing type. Things are different for arrays, however:-

```
x = dimensions(5)          # Create an array with 5 elements
LOOP I FROM 1 TO 5   # Start a loop, won't get prompt back until you type
ENDLOOP
  x[I] = I                 # Array is filled with integers, fixing its type
ENDLOOP
printn x                   # Look at the array
```

Now try assigning a real number to one of the elements of the array:-
x[1] = 3.4
You get an (almost incomprehensible !) error. The array type is fixed once an element is assigned. An array cannot hold more than one data type. However, there is nothing to stop you overwriting the whole array with :-

x = 6
printn x

x is now an integer again.

Operations

Genie can be used as a glorified calculator. For example:-

printn 2*3

Combining this with the idea of variables gives:-

x = 2; y = 3; printn x*y
hard = 2; ahead = 3; printn hard * ahead
hard = 2; ahead = 3; printn hard * ahead * 1.0
hard = 2; ahead = 3.0; printn hard * ahead
hard = 2.0 ; ahead = 3.0; printn hard * ahead

Note that you get either an integer or a real back, depending upon the types of variable that you multiply. You can use built in maths functions too:-

printn sin(3.14159265357) # value returned by sin is printed
sin_pi = sin(3.14159265357) # values stored for subsequent use
printn sin_pi

Procedures and argument passing

If you are used to Fortran, you will be used to the idea that variables passed into subroutines are passed by reference i.e. if in a subroutine you change a variable that was passed to that subroutine, then the change will be passed back out into the main program when the subroutine returns. This is ***not*** the case with simple Genie variables (integers, reals and strings), which are passed by value i.e. the procedure (subroutine) receives a copy of variables rather than the variables themselves. Any changes made inside the subroutine are not propagated back out into the main program. In this respect, GCL is like the C language. Try the following tests which use the **byval**, **byval2** and **byref** demo procedures from the tutorial file that you loaded.. You will need to look at the gcl_tutorial_ks.gcl listing to follow what is happening at each stage. First we try passing in literals.

byval # no args supplied

```
byval 1 2.0 “hello”          # int, real and string supplied
byval 1 2.0                  # int, real supplied, last value missed out
byval _ 2.0 “hello”          # do not supply a value for the first parameter, but
ensure
                             # that the correct number of arguments is supplied by
use
                             # of the ‘_’ character
```

Note that when you are passing arguments to a procedure, you separate them by whitespace and not commas.

Now pass in some variables

```
a = 1
b = 2.0
c = “hello”
byval a b c

byval2                       # changes its arguments to 5,10. and “goodbye”
byval2 1 2.0 “hello”

a = 1
b = 2.0
c = “hello”
byval2 a b c
printn “ a= “ a “ b= “ b “ c= “ c
```

You should have found that the variables a,b and c are left unaltered by **byval2**.

Like in C, arrays are passed by reference. Try

```
y=dimensions(10)
LOOP I FROM 1 TO 10
 y[I]=I*1.0
ENDLOOP
printn y
byref y
printn y
```

The procedure **byref** will have changed the contents of the y array. Workspaces (and hence all of their contents) are also passed by reference. We will deal with them later.

Multi line statements

So far we have encountered one multi line statement, the LOOP statement. We’ll now look at the others. First of all the IF statement, an example of which is contained in the procedure **ifproc**.
Try

```
ifproc 1 1 2 2
ifproc 1 2 3 4
```

ifproc 1 1 3 4

The workings of the IF statement are obvious. Not so familiar to the Fortran user is the CASE statement, which allows you to branch depending on the value of an expression. An example is contained in the procedure **caseproc** which should make the way that it operates clear. Try:-

caseproc “X”
caseproc “Y”
caseproc “grief”

Case is frequently used when processing user supplied responses to questions e.g.

```
# This is for information only, don’t bother typing it in !!!!!!!!!!!!!!!!!!
CASE ans
  IS “Y”
    # do something
  IS “N”
    # do something else
  OTHERWISE
    # chide the user and prompt for correct input
ENDCASE
```

The LOOP statement is like a Fortran DO statement. The EXITIF statement allows a conditional exit from the loop. We have already used LOOP statements that are analogous to DO statements in Fortran. We can emulate a C type “do....while” construct by setting up a loop that will run almost forever unless a certain condition is met. For example, suppose we wanted the following:-

prompt user for integer numbers until he enters a (-) number

The demo procedure **dowhile** shows how to do this. Just type ‘dowhile’.

Procedures and variable scope

Variables used within procedures *must* be declared, with the exception of LOOP counter variables which never need to be declared anywhere. Variables used only within the procedure should be declared as LOCAL. If you wish to access and use a variable value within a procedure, you have a choice:-

- pass a copy of the variable in via the parameter list
- declare the variable to be GLOBAL inside the procedure, in which case the variable may be used as if it had been passed into the procedure by *reference*. i.e. any changes you make to it will be visible outside the procedure and in any other procedure that declares that same variable name as a global.

Check out the demo procedure **scope** by typing the following commands:-

xvalue = 50

```
yvalue = 45
scope           # prints yvalue of 45 because yvalue is declared GLOBAL
                # prints xvalue of 123.45 because the xvalue declared LOCAL in
                # scope is a different variable from the xvalue outside the procedure
printn xvalue   # confirm that outside of scope, xvalue is still 50
printn yvalue   # the change made to yvalue by scope is reflected outside of scope
```

Obviously, GLOBAL variables are very convenient but they should be used with care because changes made in one procedure may produce unwanted results in another.

Procedures and parameter types

The *types* of parameters to procedures may be specified when they are declared
e.g. PARAMETERS Par1 = Real Par2 = Integer

If this is done, any attempt to supply an incorrect data type as a parameter will be greeted by an error. If the parameters are left untyped e.g. PARAMETERS Par1 Par2
then they will take their type from the parameters supplied to the procedure. This can be useful if you are writing a procedure which, for example, accepts integer or real parameters. Examples of both styles are found in the demo procedures **multiply** and **power**, which will be run shortly.

Procedures and return values

We have seen that we can get information out of a procedure using arrays, workspaces and global variables. However, procedures can also return items such as arrays, variables and workspaces. They do so when they are called as *functions*, but the syntax required when they are called as functions differs from what we have seen with our previous encounters with procedures. Look at the **multiply** demo procedure and try the following

```
multiply 4. 5.          # Result of multiplication is lost
printn multiply(4.,5.)  # Result of multiplication is printed.  Note the function style
                        # invokation of multiply  i.e the brackets and the comma
                        # separated list or parameters
x = multiply(4.,6.) ; printn multiply
answer = multiply(4.,6) # 2nd parameter is integer, routine will fail - it expects a real
```

Procedures and qualifiers

Qualifiers to procedure names allow us to modify the behaviour of a procedure. They are very much like the qualifiers seen in VMS commands e.g. dir/full, where /full is a qualifier to the dir command. An example is given in the demo procedure **power**.
Try

```
power 4
power/square 4
```

power/cube 4.

This is a useful demo because it also shows that **power** will accept either an integer or a real parameter, as the procedure does not define the type of the parameter explicitly. **power** tells us what value will be returned, but how do we access that return value ? Easy, just switch to calling **power** as a function like we did with **multiply**. However, we now have qualifiers to worry about. Try

```
res = power(4.0) ; printn res
res = power:square(4) ; printn res
```

All that we have done is replaced the '/' character with a ':' when using **power** as a function.

Creating a workspace

A workspace is very much like a C structure. Think of it as a named collection of items, a bit like a Fortran common block. A workspace is created with the fields() command. Try

```
w1 = fields()
printn w1
```

The workspace is currently empty. Add an Integer entry with

```
w1.myvalue = 10
printn w1
```

There is now an entry. Workspaces are incredibly flexible. They can contain variables, arrays and even other workspaces. For example, try

```
w1.secret = fields()
printn w1
w1.secret.message =  "No way \!"      # We must escape the '!' character
printn w1
```

The demo procedure **create_workspace** steps you through the creation of a demo workspace, showing you each line before it is executed. The procedure itself returns the created workspace. Try typing:-

```
raspberry = create_workspace()        # answer the questions
printn raspberry
```

Workspaces may be passed to other procedures. They pass by reference as do all of their contents i.e. all of their contents are available for modification. Check out the demo procedure **takeover** which accepts the recently created 'raspberry' workspace and alters its contents. It also returns the modified workspace. Try

```
isis = takeover(raspberry)
printn isis
printn raspberry
```

The ‘isis’ and ‘raspberry’ workspaces are identical.

A practical example of workspace usage and array handling

Read in a .PRO file created by the CCSL program sr15ls. There are 5001 rows in the data file, with 5 columns of data. We can read in this data using the read/ascii/columns command which, when called as a function, returns a workspace populated with the data.
Try typing:-

```
data = read:ascii:columns(“CT130K.PRO”,”x,background,meas,calc,err”)
printn data
```

The data workspace contains 5 arrays with names as supplied to the read function. For plotting purposes, we want a difference divided by the esd on the calculated values. This can be done with a single line in Genie i.e. no need to loop through the 5001 points.
Try typing:-

```
data.doesd = (d.meas - d.calc) / d.err
```

Now we are ready to plot. Type:-

```
device/open
plotit data                # plotit is a simple demo procedure to put up a fit and diff. plot
```

Listing of gcl_tutorial_ks.gcl file

```
PROCEDURE byval
#
# Show how parameters are passed into a procedure
#
PARAMETERS X = Integer Y = Real Z = String
printn "In procedure, x = " x ", y = " y ", z = " z
ENDPROCEDURE

PROCEDURE byval2
#
# Show how parameter variables altered inside a procedure do not
# affect the variables outside the procedure itself
#
PARAMETERS X = Integer Y = Real Z = String
printn "Supplied values: x = " x ", y = " y ", z = " z
printn "Changing values to 5,10.0,\"goodbye\""
x=5 ; y=10. ; z="goodbye"
printn "Modified values: x = " x ", y = " y ", z = " z
ENDPROCEDURE

PROCEDURE byref
#
# Show how array parameters altered inside a procedure are
# affected outside the procedure itself
#
PARAMETERS X = RealArray
LOCAL SIZE              # tmp variable only used inside procedure
SIZE = LENGTH(X)        # length() returns the size of the array
LOOP I FROM 1 TO SIZE
  X[I]=I*2.0                # multiply supplied array elements by
2,
ENDLOOP                     # but note that we could have achieved
the
ENDPROCEDURE                # same result by x=x*2.0

PROCEDURE ifproc
#
# Shows the mechanics of an if statement
#
PARAMETERS One = Integer Two = Integer Three = Integer Four = Integer
IF (ONE = TWO) AND (THREE = FOUR)
  printn "Both sets equal"
  ELSEIF (ONE = TWO)
  printn "First set equal"
ELSEIF (THREE = FOUR)
  printn "Second set equal"
ELSE
  printn "Neither set equal"
ENDIF
ENDPROCEDURE




PROCEDURE caseproc
#
# Shows the mechanics of a case statement
#
```

```
PARAMETERS casevar = String
CASE casevar
 IS "X"
   printn "You gave me an X"
 IS "Y"
   printn "You gave me a Y"
 OTHERWISE
   printn "You gave me " casevar " !"
ENDCASE
ENDPROCEDURE

PROCEDURE dowhile
#
# Emulate a "do-while" loop
#
LOCAL ANS SUM=0               # Sum is initialised to zero
LOOP I FROM 1 TO 10000 STEP 1 # Effectively an infinite loop
  ANS=INQUIRE("Enter an integer value, (-) integer to finish > ")
  EXITIF (ANS < 0)
  SUM = SUM + ANS
ENDLOOP
PRINTN "The sum of the numbers entered is " sum
ENDPROCEDURE

PROCEDURE scope
#
# Show difference between local and global variables
#
GLOBAL YVALUE           # Gives access to any existing 'yvalue'
variable
LOCAL XVALUE = 123.45   # Different from any other 'xvalue' variable
printn "In procedure, yvalue is " yvalue
printn "In procedure, xvalue is " xvalue
yvalue = 999.9
ENDPROCEDURE


PROCEDURE multiply
#
# Shows how a function can return a value
#
PARAMETERS First = Real Second = Real
RESULT Product
Product = First * Second
ENDPROCEDURE









PROCEDURE power
#
# Show operation of procedure qualifiers
#
QUALIFIERS /SQUARE /CUBE
PARAMETERS Argument
RESULT answer
Argument = Argument * 1.0     # Ensure that Argument is a real value
IF SQUARE
```

```
  answer = argument^2
ELSEIF CUBE
  answer = argument^3
ELSE
  answer = argument
ENDIF
printn "Procedure will return value " answer
ENDPROCEDURE

PROCEDURE create_workspace
#
# Creates a workspace, populates it, returns it
#
RESULT wc
LOCAL hero
hero = inquire("wc=fields()")
wc=fields()
hero = inquire("wc.xlabel=\"Two theta\"")
wc.xlabel="Two theta"
hero = inquire("wc.ylabel=\"Counts\"")
wc.ylabel="Counts"
hero = inquire("wc.title=\"Typical Daresbury background\"")
wc.title="Typical Daresbury background"
hero = inquire("wc.data=dimensions(100)")
wc.data=dimensions(100)
hero = inquire("LOOP I FROM 1 TO 100;wc.data[I]=9999.0;ENDLOOP")
LOOP I FROM 1 TO 100
  wc.data[I] = 9999.0         # NB: Could have used FILL command !
ENDLOOP
ENDPROCEDURE

PROCEDURE takeover
#
# Takes the array returned by create_workspace and effects some
# changes.  Returns a copy of the modified workspace.
#
PARAMETERS arg1 = Workspace
RESULT wc
LOCAL hero
hero = inquire("arg1.xlabel=\"TOF\"")
arg1.xlabel="TOF"
hero = inquire("arg1.ylabel=\"Counts\"")
arg1.ylabel="Counts"
hero = inquire("arg1.title=\"Typical ISIS background\"")
arg1.title="Typical ISIS background"
hero = inquire("LOOP I FROM 1 TO 100;arg1.data[I]=0.0;ENDLOOP")
LOOP I FROM 1 TO 100
  arg1.data[I] = 0.0
ENDLOOP
wc = arg1
ENDPROCEDURE

PROCEDURE plotit
#
# Rapid plot for Genie demo
#
PARAMETERS D
WINDOW/AUTOSCALED .2 .9 .4 .9 D.x D.meas
DRAW/PLOT D.x D.calc 1 1 2
DRAW/MARKERS D.x D.meas 1 20 1.2 2
DRAW/AXES 1 2 1
DRAW/LABELS "2\\gh / degrees" "Counts" 1 1.4 _ 2
WINDOW/AUTOSCALED .2 .9 .2 .25 D.x D.doesd
DRAW/PLOT D.x D.doesd 1 1 2
DRAW/AXES/YHORIZONTAL 1 2 .7
DRAW/LABELS " " "Diff/esd" 1 1.2 _ 2
```

```
ENDPROCEDURE
```