# GNU Bayonne Script Examples

Luca Bariani
LucaBariani@Ferrara.Linux.it

May 2003

GNU Bayonne
http://www.gnu.org/software/bayonne

Ferrara Linux User Group
http://Ferrara.Linux.it

# Contents

# List of Examples

# 1 Introduction

GNU Bayonne is a script driven telephony application server. As such, it has it's own scripting language built by class extension from the GNU ccScript interpreter. The main characteristics of the ccScript scripting language are described in the document: **GNU Bayonne Script Programming Guide**; this document gives a lot of examples to show how programming Bayonne using ccScript.

Every example of this document is tested in the indicated environment (Bayonne version, ccScript version, ...) and the output reported after the source.

This document will never be complete: there are always new examples to add (by adding new feature og changing it).

# 2 Before you begin

## 2.1 Testing

Every example of this document has been tested with Ccscript and/or Bayonne version's as reported in the *Testing environment* line. To run example *example.scr* and get the output *example.out* as reported here follow this steps:

- Testing environment: Ccscript (only):
  - `ccscript example.scr`
- Testing environment: Ccscript, Bayonne 1.0.x:
  - start Bayonne 1.0.x with Dummy driver from a root shell with
    `bayonne_start -x -driver dummy`
  - start example from another shell with
    `bayonne_control start 0 example`
    you'll see output in the Bayonne starting shell.
    Note 1: the script *example.scr* must be located in the directory */usr/-share/ccScript* (or in the path indicated by *bayonne.conf* configuration file).
    Note 2: when Bayonne server is running, after every change on *example.scr* the server needs to recompile the script with the command `bayonne_control compile`.
- Testing environment: Ccscript, Bayonne 1.2.x:
  - start Bayonne 1.2.x with Dummy driver from a root shell with
    `bayonne -x --driver dummy`
  - start example from another shell with
    `bayonne --control start 0 example`
    you'll see output in the Bayonne starting shell.
    Note 1: the script *example.scr* must be located in the directory */usr/-local/share/bayonne* (or in the path indicated by *bayonne.conf* configuration file).
    Note 2: when Bayonne server is running, after every change on *example.scr* the server needs to recompile the script with the command `bayonne --control compile`.

## 2.2 Dummy Driver

The Dummy driver simulates IVR hardware behavior using a normal sound card and a keyboard, so it's possible to run Bayonne without the specific IVR hardware; it's very useful to make a testing/developing environment or to try Bayonne itself. Using Dummy driver is very simple:

- voice input and audio output work through sound card;
- the phone keyboard works through PC keyboard (key: 0...9, * and #);
- an incoming call begins through the 'space' key;
- a call ends through the 'H' key (hang up).
- dial tone through the 'D' key (dialtone).
- busy tone through the 'B' key (busytone).
- exit 'X' key (exit).

So a phone call simulation is very easy:

- call start through 'space' key;
- call progress: voice input and output through audio card, numeric input through keyboard (key: 0...9, * and #);
- call end through 'H' key.

## 2.3 Right ending

Every Bayonne script program must end with the *exit* instruction, so everything is closed in the right mode. Without the *exit* instruction, a Bayonne script has an infinite loop. With Ccscript only (without Bayonne) a script can end without the *exit* instruction without any error. The first two examples are equal, but have different *testing environment*, so they have completely different output and behavior.

### Example 1: exit, ccscript

*Testing environment: Ccscript 2.5.1*

**Source: luca-noexit1.scr**

```
slog "No exit at the end: with ccscript no problem"
```

**Output: luca-noexit1.out**

```
luca-noexit1: 1 steps compiled
luca-noexit1: No exit at the end: with ccscript no problem
exiting..
```

The output is three lines only, the first is the compile report, the last is for exit and the second is the real output of the script.

### Example 2: exit, bayonne

*Testing environment: Ccscript 2.5.1, Bayonne 1.0.10, Bayonne 1.2.4*

**Source: luca-noexit2.scr**

```
slog "No exit at the end: infinite loop"
```

**Output: luca-noexit2.out**

```
dummy(0): luca-noexit2: No exit at the end: infinite loop
dummy(0): luca-noexit2: No exit at the end: infinite loop
dummy(0): luca-noexit2: No exit at the end: infinite loop
dummy(0): luca-noexit2: No exit at the end: infinite loop
dummy(0): luca-noexit2: No exit at the end: infinite loop
dummy(0): luca-noexit2: No exit at the end: infinite loop
dummy(0): luca-noexit2: No exit at the end: infinite loop
dummy(0): luca-noexit2: No exit at the end: infinite loop
dummy(0): luca-noexit2: No exit at the end: infinite loop
dummy0: hangup...
```

The output is an infinite loop, to end this we must press 'H' and terminate the call.

# 3 Variables

The basic use of Ccscript variables needs the *set* instruction (assignment, concatenation); to delete a variable value there is the *clear* instruction. The instruction *slog* is very useful for logging out messages.

## 3.1 Definition and use

A basic variable is defined with the *set* instruction; an undefined variable is like an empty one.

### Example 3: set, clear, slog

*Testing environment: Ccscript 2.5.1*

**Source: luca-set1.scr**

```
slog %variable          # empty value
set %variable "luca"    # new value
slog %variable          # prints "luca"
clear %variable         # empty again
slog %variable          # no output
exit                    # NB: always an exit at end
```

**Output: luca-set1.out**

```
luca-set1: 6 steps compiled
luca-set1:
luca-set1: luca
luca-set1:
exiting...
```

The second line is empty because the variable is undefined, the forth line is empty because variable has been cleared.

The *set* instruction is used in assignment, but in some cases for this purpose the symbol '=' can be used. Using multiple values with *set* instruction we got string concatenation.

## Example 4: assignment, concatenation

*Testing environment: Ccscript 2.5.1*

**Source: luca-set2.scr**

```
set %var1 "Luca"        # string assignment
slog %var1              # prints "Luca"
set  %var2 "10"         # numeric assignment
slog %var2              # prints "10"
%var3 = 100             # alternative numeric assignment
slog %var3              # prints "100"
%var4 = "Luke"          # WRONG: only numbers
slog %var4              # prints "0" and not the "Luke"
set %var5="Bariani"     # both "set" and "="
slog %var5

set %concat1 %var1 %var2 %var3  # string concatenation
slog %concat1                   # prints "Luca10100"
set %concat2 %var1, " ", %var5
   # string concatenation with a costant
   # using comma or not is indifferent
slog %concat2                   # prints "Luca Bariani"
slog %var1 " " %var5            # slog string concatenation
                                # prints "Luca Bariani" again
exit                    # NB: always an exit at the end
```

**Output: luca-set2.out**

```
luca-set2: 16 steps compiled
luca-set2: Luca
luca-set2: 10
luca-set2: 100
luca-set2: 0
luca-set2: Bariani
luca-set2: Luca10100
luca-set2: Luca Bariani
luca-set2: Luca Bariani
exiting...
```

The instructions *set.min* and *set.max* allow variable assignment/definition getting the minimum/maximum value from the specified list. The instruction *swap* allows setting two variables to each other's value.

## Example 5: min, max, swap

*Testing environment: Ccscript 2.5.1*

**Source: luca-set3.scr**

```
set %var1=1             # numeric value
set %var2=10            # numeric value
set %var3=100           # numeric value

slog "var1="%var1 ", var2=" %var2 ",var3="  %var3

set.min %min %var1 %var2 %var3  # set to the lesser value
set.max %max %var1 %var2 %var3  # set to the greater values

slog "min: " %min " max: " %max

swap %var1 %var3        # variable exchange
slog "var1="%var1 ", var2=" %var2 ",var3="  %var3

exit                    # NB: always an exit at the end
```

**Output: luca-set3.out**

```
luca-set3: 10 steps compiled
luca-set3: var1=1, var2=10,var3=100
luca-set3: min: 1 max: 100
luca-set3: var1=100, var2=10,var3=1
exiting...
```

## 3.2 Basic numerical operations

The instructions *inc* and *dec* perform the basic numeric operations.

## Example 6: inc, dec, sum, difference

*Testing environment: Ccscript 2.5.1*

**Source: luca-inc.scr**

```
set %var1=10            # numeric value
set %var2=10            # numeric value
inc %var1               # single increment
dec %var2               # single decrement
slog "var1=" %var1 ", var2="%var2
inc %var1 4             # arbitrary increment: sum
dec %var2 4             # arbitrary decrement: difference
slog "var1=" %var1 ", var2="%var2
inc %var1 %var2         # increment with no costants
slog "var1=" %var1 ", var2="%var2
exit                    # NB: always an exit at the end
```

**Output: luca-inc.out**

```
luca-inc: 11 steps compiled
luca-inc: var1=11, var2=9
luca-inc: var1=15, var2=5
luca-inc: var1=20, var2=5
exiting..
```

# 4 Packages

A Ccscript package provides new commands and properties to your script. These new features are defined in external modules. To import a package and use its functions there is the command/directive *use*, so *use mypack* imports the package mypack.

## 4.1 Digits

The *digits* package provides properties allowing manipulation of strings. While intended for strings of digits, it works equally well on generic strings.
The *trim* instruction normalizes a string to a specific length. Normalization can be based on specific offset: start, middle, end. If no offset is specified the default is start. Offset indexing begins with 0, as is the case in Python and Perl.

### Example 7: digits, trim

*Testing environment: Ccscript 2.5.1*

**Source: luca-digits1.scr**

```
use digits
set %var "1234567890"  # a generic variable
trim.start %var 6      # leave the first 6 digits
slog "var: " %var
set %var "1234567890"
trim.end %var 6        # leave the last 6 digits
slog "var: " %var
set %var "1234567890"
trim.2 %var 6          # leave the first 6 digits after
                       # the second digit
slog "var: " %var
set %var "1234567890"
trim.start %var 10     # leave the first 10 digits: %var
                       # is inaltered
slog "var: " %var
exit                # NB: always an exit at the end
```

**Output: luca-digits1.out**

```
luca-digits1: 14 steps compiled
luca-digits1: var: 123456
luca-digits1: var: 567890
luca-digits1: var: 345678
luca-digits1: var: 1234567890
exiting...
```

### Example 8: digits, trim

*Testing environment: Ccscript 2.5.1*

**Source: jim-digits1.scr**

```
# vim: set ts=8 sw=8:
use digits
set %var "1234567890"
slog "Original string:    " %var
trim.start %var 7
slog "First 7 digits:     " %var
trim.end %var 5
slog "Last 5 digits:      " %var
trim.1 %var 2
slog "2 digits, offset 1: " %var
trim.start %var 4     # var unaltered as it's only 2 digits long
slog "First 4 digits:     " %var
trim %var 1     # default offset is start
slog "First digit:        " %var
slog ""
set %var "abnormally"
slog "Works with letters also: "
slog %var
trim.start %var 8
slog %var
trim.end %var 6
slog %var
trim.1 %var 2
slog %var
trim %var 1
slog %var
exit
```

**Output: jim-digits1.out**

```
jim-digits1: 26 steps compiled
jim-digits1: Original string:    1234567890
jim-digits1: First 7 digits:     1234567
jim-digits1: Last 5 digits:      34567
jim-digits1: 2 digits, offset 1: 45
jim-digits1: First 4 digits:     45
jim-digits1: First digit:        4
jim-digits1:
jim-digits1: Works with letters also:
jim-digits1: abnormally
jim-digits1: abnormal
jim-digits1: normal
jim-digits1: or
jim-digits1: o
exiting...
```

The *chop* instruction cuts digits from a specific offset: start, middle, end. *Chop* is the inverse of *trim*. Where *trim* saves the specified part of the string, *chop* deletes it.

## Example 9: digits, chop

*Testing environment: Ccscript 2.5.1*

**Source: luca-digits2.scr**

```
use digits
set %var "1234567890"  # a generic variable
chop.start %var 6      # cuts the first 6 digits
slog "var: " %var
set %var "1234567890"
chop.end %var 6        # cuts the last 6 digits
slog "var: " %var
set %var "1234567890"
chop.2 %var 6          # cuts the first 6 digits after
                       # the second digit
slog "var: " %var
set %var "1234567890"
chop.10 %var 10        # cuts digits after the tenth digit: %var
                       # is inaltered
slog "var: " %var
exit                # NB: always an exit at the end
```

**Output: luca-digits2.out**

```
luca-digits2: 14 steps compiled
luca-digits2: var: 7890
luca-digits2: var: 1234
luca-digits2: var: 1290
luca-digits2: var: 1234567890
exiting...
```

The *delete* instruction deletes a specific digit if present in the specified position.

## Example 10: digits, delete

*Testing environment: Ccscript 2.5.2*

**Source: luca-digits3.scr**

```
use digits
set %var "abcdefg"     # a generic variable
delete.start %var "a"  # if the first digit is "a" deletes it
slog "var: " %var
set %var "abcdefg"
delete.3 %var "d"  # if the fourth digit (the first digit is
                   #  the number 0) is "d" deletes it
slog "var: " %var
set %var "abcdefg"
delete.0 %var "a"    # delete.0 is equal to delete.start
slog "var: " %var
set %var "abcdefg"
delete.2 %var "e"    # the 3rd digit in not "e", nothing happens
slog "var: " %var
set %var "abcdefg"
```

```
delete.end %var "g"  # if the last digit is "g" deletes it
slog "var: " %var
exit                # NB: always an exit at the end
```

**Output: luca-digits3.out**

```
luca-digits3: 17 steps compiled
luca-digits3: var: bcdefg
luca-digits3: var: abcefg
luca-digits3: var: bcdefg
luca-digits3: var: abcdefg
luca-digits3: var: abcdef
exiting...
```

## Example 11: digits, delete

*Testing environment: Ccscript 2.5.2*

**Source: jim-digits3.scr**

```
# vim: set ts=8 sw=8:
use digits
set %var "abnormally"
slog "Original string:         " %var
delete.start %var "ab"
slog "delete.start ab:         " %var
delete.end %var "ly"
slog "delete.end ly:           " %var
delete.2 %var "rm"
slog "delete.2 rm:             " %var
exit
```

**Output: jim-digits3.out**

```
jim-digits3: 10 steps compiled
jim-digits3: Original string:         abnormally
jim-digits3: delete.start ab:         normally
jim-digits3: delete.end ly:           normal
jim-digits3: delete.2 rm:             noal
exiting...
```

The *insert* instruction inserts a digit in a specific offset. *Insert* did not accept the *end* keyword until ccscript 2.5.2.

## Example 12: digits, insert

*Testing environment: Ccscript 2.5.1*

**Source: luca-digits4.scr**

```
use digits
set %var "123456"     # a generic variable
insert.start %var "a" # insert "a" at the beginning of %var
slog "var: " %var
set %var "123456"
insert.3 %var "AA"   # insert after the fourth digit
                     # (the first digit is the number 0)
slog "var: " %var
set %var "123456"
insert.0 %var "a"    # insert.0 is equal to insert.start
slog "var: " %var
exit                 # NB: always an exit at the end
```

**Output: luca-digits4.out**

```
luca-digits4: 11 steps compiled
luca-digits4: var: a123456
luca-digits4: var: 123AA456
luca-digits4: var: a123456
exiting...
```

## Example 13: digits, insert

*Testing environment: Ccscript 2.5.1*

**Source: jim-digits4.scr**

```
# vim: set ts=8 sw=8:
use digits
set %var "noal"
slog "Original string:         " %var
insert.2 %var "rm"
slog "Insert.2 rm:             " %var
insert.start %var "ab"
slog "insert.start ab:         " %var
insert.end %var "ly"
slog "insert.end ly:           " %var
exit
```

**Output: jim-digits4.out**

```
jim-digits4: 10 steps compiled
jim-digits4: Original string:         noal
jim-digits4: Insert.2 rm:             normal
jim-digits4: insert.start ab:         abnormal
jim-digits4: insert.end ly:           lyabnormal
exiting...
```

The *prefix* instruction inserts a digit in at a beginning of a string, but only if it's not already present. Using an offset or the *end* keyword leaves the string unchanged.

## Example 14: digits, prefix

*Testing environment: Ccscript 2.5.1*

**Source: luca-digits5.scr**

```
use digits
set %var "123456"      # a generic variable
prefix.start %var "a"  # insert "a" at the beginning of
                       # %var because $var starts with 1
slog "var: " %var
set %var "123456"
prefix.0 %var "a"     # insert.0 is equal to insert.start
slog "var: " %var
set %var "123456"
prefix.start %var "1"  # doesn't insert "1" at the beginning
                       # of %var because $var already
       # starts with 1
slog "var: " %var
exit                 # NB: always an exit at the end
```

**Output: luca-digits5.out**

```
luca-digits5: 11 steps compiled
luca-digits5: var: a123456
luca-digits5: var: a123456
luca-digits5: var: 123456
exiting...
```

## Example 15: digits, prefix

*Testing environment: Ccscript 2.5.1*

**Source: jim-digits5.scr**

```
# vim: set ts=8 sw=8:
use digits
set %var "normal"
slog "Original string:               " %var
prefix.start %var "no"
slog "prefix.start 'no':             " %var
prefix.start %var "ab"
slog "prefix.start 'ab':             " %var
prefix.end %var "ly"
slog "prefix.end 'ly' does nothing:  " %var
prefix.4 %var "licious"
slog "Nor does prefix.4:             " %var
exit
```

**Output: jim-digits5.out**

```
jim-digits5: 12 steps compiled
jim-digits5: Original string:              normal
jim-digits5: prefix.start 'no':            normal
```

```
jim-digits5: prefix.start 'ab':             abnormal
jim-digits5: prefix.end 'ly' does nothing:  lyabnormal
jim-digits5: Nor does prefix.4:             lyabnormal
exiting...
```

The *replace* instruction replace a character with different string, at the specific offset.

## Example 16: digits, replace

*Testing environment: Ccscript 2.5.1*

**Source: luca-digits6.scr**

```
use digits
set %var "abcdefg"          # a generic variable
replace.start %var "a" "A" # if the first digit is "a"
                           # replaces it
slog "var: " %var
set %var "abcdefg"
replace.3 %var "d" "D" # if the fourth digit (the first digit
                       # is the number 0) is "d" replaces it
slog "var: " %var
set %var "abcdefg"
replace.0 %var "a" "A" # replace.0 is equal to replace.start
slog "var: " %var
set %var "abcdefg"
replace.2 %var "e" "E"    # the 3rd digit in not "e"
                          # nothing happens
slog "var: " %var
set %var "abcdefg"
replace.end %var "g" "G"    # if the last digit is "g"
                            # replaces it
slog "var: " %var
exit                # NB: always an exit at the end
```

**Output: luca-digits6.out**

```
luca-digits6: 17 steps compiled
luca-digits6: var: Abcdefg
luca-digits6: var: abcDefg
luca-digits6: var: Abcdefg
luca-digits6: var: abcdefg
luca-digits6: var: abcdefG
exiting...
```

## Example 17: digits, replace

*Testing environment: Ccscript 2.5.1*

**Source: jim-digits6.scr**

```
# vim: set ts=8 sw=8:
use digits
set %var "abnormally"
slog "Original string:          " %var
replace.2 %var "normal" "ysmal"
slog "Updated string:           " %var
exit
```

**Output: jim-digits6.out**

```
jim-digits6: 6 steps compiled
jim-digits6: Original string:          abnormally
jim-digits6: Updated string:           abysmally
exiting...
```

Replacing a character with another string:

## Example 18: digits, replace/2

*Testing environment: Ccscript 2.5.1*

**Source: luca-digits7.scr**

```
use digits
set %var "abcdefg"           # a generic variable
replace.start %var "a" "AB" # if the first digit is "a"
                             # replaces it with AB
slog "var: " %var
set %var "abcdefg"
replace.end %var "g" "GH"   # if the last digit is "g"
                             # replaces it with GH
slog "var: " %var
exit                # NB: always an exit at the end
```

**Output: luca-digits7.out**

```
luca-digits7: 8 steps compiled
luca-digits7: var: ABbcdefg
luca-digits7: var: abcdefGH
exiting...
```

## 4.2 Random

The *random* package provides a flexible random number generator (RNG). The RNG may be seeded with a known value at any time. If no seed is used the current time of day is used.

## Example 19: random

*Testing environment: Ccscript 2.5.1*

**Source: jim-random1.scr**

```
# vim: set ts=8 sw=8:
use random
random.seed 6 # same numbers each time we run
repeat 5
random.9 %var # 1 <= %var <= 9
slog %var
loop
slog " "
random.seed 6 # Restart RNG
repeat 5
random.9 %var # Same numbers as above
slog %var
loop
slog " "
random.9 seed=6 %var1 %var2 %var3 %var4 %var5
# Same numbers, once again
slog %var1 ", " %var2 ", " %var3 ", " %var4 ", " %var5
exit
```

**Output: jim-random1.out**

```
jim-random1: 16 steps compiled
jim-random1: 2
jim-random1: 9
jim-random1: 8
jim-random1: 4
jim-random1: 1
jim-random1:
jim-random1: 2
jim-random1: 9
jim-random1: 8
jim-random1: 4
jim-random1: 1
jim-random1:
jim-random1: 2, 9, 8, 4, 1
exiting...
```

There are several modifiers for the random number stream.

**seed:** It is possible to reseed the RNG every time it is used. This is usually A Bad Idea (tm).

**count:** Generate multiple random numbers, summing them together. An example of where this is useful is in simulating rolls of dice. For example, **random.6 count=2** returns a value from 2 to 12 inclusive, simulating the roll of two 6 sided dice.

**offset:** Normally the RNG generates numbers ¿= 1. This can be changed by with the offset. For example, **random.6 offset=3** generates a random number from 4 to 9 inclusive.

**min:** Ensures the number generated is at least the minimum value. If the number is < min it is set to min.

**max:** Ensures the number generated is less than or equal to max. If the number is > max it is set to max.

**reroll:** Once the min, max, offset, and count are factored in, if the resulting value is less than the reroll value then a new number is generated. Hence it's possible to let the user roll 2 6 sided dir until the total is 5 or more.

## Example 20: random

*Testing environment: Ccscript 2.5.1*

**Source: jim-random2.scr**

```
# vim: set ts=8 sw=8:
use random
repeat 20
random.6 count=2 %var1 # Simulate throwing 2 dice
slog %var1
loop
slog " "
repeat 20
random.6 count=2 reroll=4 %var1 # Simulate throwing 2 dice
slog %var1
loop
exit
```

**Output: jim-random2.out**

```
jim-random2: 11 steps compiled
jim-random2: 8
jim-random2: 5
jim-random2: 7
jim-random2: 5
jim-random2: 4
jim-random2: 10
jim-random2: 6
jim-random2: 6
jim-random2: 11
jim-random2: 6
jim-random2: 7
jim-random2: 8
jim-random2: 6
jim-random2: 6
jim-random2: 2
jim-random2: 11
jim-random2: 8
jim-random2: 6
jim-random2: 6
jim-random2: 5
jim-random2:
jim-random2: 6
jim-random2: 10
jim-random2: 7
jim-random2: 5
```

```
jim-random2: 6
jim-random2: 12
jim-random2: 11
jim-random2: 9
jim-random2: 6
jim-random2: 8
jim-random2: 6
jim-random2: 10
jim-random2: 12
jim-random2: 7
jim-random2: 6
jim-random2: 6
jim-random2: 8
jim-random2: 5
jim-random2: 5
jim-random2: 6
exiting...
```

Notice the way min and max are defined. As can be seen in the following example, careless use weights the RNG stream to the minimum/maximum, which is probably not what is wanted. Use a range and an offset to get a true set of random numbers in a given range.

## Example 21: random

*Testing environment: Ccscript 2.5.1*

**Source: jim-random3.scr**

```
# vim: set ts=8 sw=8:
use random
random.seed 300
random.4 seed=0 offset=4 %var1 %var2 %var3 %var4 %var5 \
%var6 %var7 %var8 %var9 %var10 %var11 %var12   \
%var13 %var14 %var15 %var16 %var17 %var18      \
%var19 %var20
slog "Evenly distributed: "
slog %var1 ", " %var2 ", " %var3 ", " %var4 ", "    \
%var5 ", " %var6 ", " %var7 ", " %var8 ", "    \
%var9 ", " %var10 ", "%var11 ", " %var12 ", "  \
%var13 ", " %var14 ", " %var15  ", " %var16    \
", " %var17 ", " %var18 ", " %var19 ", " %var20

random.10 seed=0 min=5 max=8 %var1 %var2 %var3 %var4   \
%var5 %var6 %var7 %var8 %var9 %var10 %var11    \
%var12 %var13 %var14 %var15 %var16 %var17      \
%var18 %var19 %var20
slog "Weighted at 5 and 8:"
slog %var1 ", " %var2 ", " %var3 ", " %var4 ", "    \
%var5 ", " %var6 ", " %var7 ", " %var8 ", "    \
%var9 ", " %var10 ", "%var11 ", " %var12 ", "  \
%var13 ", " %var14 ", " %var15  ", " %var16    \
", " %var17 ", " %var18 ", " %var19 ", " %var20
```

```
random.4 seed=0 offset=4 %var1 %var2 %var3 %var4 %var5 \
%var6 %var7 %var8 %var9 %var10 %var11 %var12   \
%var13 %var14 %var15 %var16 %var17 %var18      \
%var19 %var20
exit
```

**Output: jim-random3.out**

```
jim-random3: 10 steps compiled
jim-random3: Evenly distributed:
jim-random3: 8, 6, 8, 8, 8, 5, 6, 8, 6, 7, 6, 7, 6, 7, 8, 8,
7, 7, 5, 7
jim-random3: Weighted at 5 and 8:
jim-random3: 8, 5, 8, 8, 8, 5, 5, 8, 5, 6, 5, 7, 5, 6, 8, 8,
7, 8, 5, 7
exiting...
```

## 4.3 Sort

The *sort* package allows sorting data structures such as a fifo, stack, list,...

### Example 22: sort, sequence

*Testing environment: Ccscript 2.5.1*

**Source: luca-sort1.scr**

```
use sort
sequence 5 %seq
post %seq 3 5 2 1 4    # five values in random order
dup %seq %seqUp        # sequenze duplication
dup %seq %seqDown      #
sort %seqUp            # normal sort
sort.reverse %seqDown  # reverse sort
slog "original: " %seq " up: " %seqUp " down: " %seqDown
slog "original: " %seq " up: " %seqUp " down: " %seqDown
slog "original: " %seq " up: " %seqUp " down: " %seqDown
slog "original: " %seq " up: " %seqUp " down: " %seqDown
slog "original: " %seq " up: " %seqUp " down: " %seqDown
exit                    # NB: always an exit at end
```

**Output: luca-sort1.out**

```
luca-sort1: 13 steps compiled
luca-sort1: original: 3 up: 1 down: 5
luca-sort1: original: 5 up: 2 down: 4
luca-sort1: original: 2 up: 3 down: 3
luca-sort1: original: 1 up: 4 down: 2
luca-sort1: original: 4 up: 5 down: 1
exiting...
```

### Example 23: sort, list

*Testing environment: Ccscript 2.5.1*

**Source: luca-sort2.scr**

```
use sort
set %list "2,5,3,4,1"
dup %list %listUp        # list duplication
dup %list %listDown      #
sort %listUp             # normal sort
sort.reverse %listDown   # reverse sort
slog "original:     "%list
slog "up ordered:   "%listUp
slog "down ordered: "%listDown
exit                     # NB: always an exit at end
```

**Output: luca-sort2.out**

```
luca-sort2: 10 steps compiled
luca-sort2: original:     2,5,3,4,1
luca-sort2: up ordered:   1,2,3,4,5
luca-sort2: down ordered: 5,4,3,2,1
exiting...
```

# 5 Data structures

Ccscript allows some complex data structures and some specific and automatic operations on them.

## 5.1 Time and Date

Ccscript allows time and date manipulation with particular data structures. For definition and assignment there are instructions *set.time* and *set.date*. These data structures support basic operation like *inc* and *dec* which results are always valid time or date values.

### Example 24: date

*Testing environment: Ccscript 2.5.1*

**Source: luca-date.scr**

```
use date
set.date %date "20031225"      # a date
dup %date %minus1              # date copy
dup %date %minus30
dup %date %add1
dup %date %add10
slog "date (mouth/day/year): "%date
dec.date %minus1         # date - 1 day
dec.date %minus30 30     # date - 30 days
```

```
inc.date %add1           # date + 1 day
inc.date %add10 10       # date + 10 days
slog "one day before "%minus1
slog "30 days before "%minus30
slog "one day after  "%add1
slog "10 days after  "%add10
exit                      # NB: always an exit at the end
```

**Output: luca-date.out**

```
luca-date: 16 steps compiled
luca-date: date (mouth/day/year): 12/25/2003
luca-date: one day before 12/24/2003
luca-date: 30 days before 11/25/2003
luca-date: one day after  12/26/2003
luca-date: 10 days after  01/04/2004
exiting...
```

## Example 25: time

*Testing environment: Ccscript 2.5.1*

**Source: luca-time.scr**

```
use time
set.time %time "115700"           # a time
dup %time %minus1
dup %time %minus70
dup %time %minus4000
dup %time %add1
dup %time %add70
dup %time %add4000
slog "time (HH:MM:SS): "%time
dec.time %minus1
slog "one second before: " %minus1
dec.time %minus70 70
slog "70 seconds before: " %minus70
dec.time %minus4000 4000
slog "4000 seconds before: " %minus4000
inc.time %add1
slog "one second after: " %add1
inc.time %add70 70
slog "70 seconds after: " %add70
inc.time %add4000 4000
slog "4000 seconds after: " %add4000
exit                # NB: always an exit at the end
```

**Output: luca-time.out**

```
luca-time: 22 steps compiled
luca-time: time (HH:MM:SS): 11:57:00
luca-time: one second before: 11:56:59
luca-time: 70 seconds before: 11:55:50
```

```
luca-time: 4000 seconds before: 10:50:20
luca-time: one second after: 11:57:01
luca-time: 70 seconds after: 11:58:10
luca-time: 4000 seconds after: 13:03:40
exiting...
```

## 5.2 Counter

A *counter* data structure is a variable which is incremented at every use.

### Example 26: counter

*Testing environment: Ccscript 2.5.1*

**Source: luca-count.scr**

```
counter %count          # counter definition
slog "counter " %count  # prints count and increments it
slog "counter " %count  # prints count and increments it
slog "counter " %count  # prints count and increments it
exit                    # NB: always an exit at the end
```

**Output: luca-count.out**

```
luca-count: 5 steps compiled
luca-count: counter 1
luca-count: counter 2
luca-count: counter 3
exiting...
```

## 5.3 Dynamic structures: Fifo, Stack, Sequence

The *stack* and *fifo* are dynamic structures (FIFO = First In First Out, Stack = LIFO = Last In First Out); they store values with *post* instruction and release them at every use, a value can be deleted with *remove*. Of course using *post* and *remove* don't release a value from structure.

### Example 27: stack

*Testing environment: Ccscript 2.5.1*

**Source: luca-stack.scr**

```
stack 3 %stck        # stack declaration
post %stck 1         # insert a value, stck = 1
post %stck 2         # insert a value, stck = 2 1
post %stck 3         # insert a value, stck = 3 2 1
slog "stck " %stck
     # prints the stack value and remove it, stck = 2 1
slog "stck " %stck
     # prints the stack value and remove it, stck = 1
slog "stck " %stck
     # prints the stack value and remove it, stck =
```

```
slog "stck " %stck
     # prints an empty value: the stack is empty

post %stck 1 2 3     # multiple post, stck = 3 2 1
slog "stck " %stck
     # prints the stack value and remove it, stck = 2 1
slog "stck " %stck
     # prints the stack value and remove it, stck = 1
slog "stck " %stck
     # prints the stack value and remove it, stck =

post %stck 1 2 3 4   # too many elements: the stack has size 3
                     # the fourth value is ignored
slog "stck " %stck   # prints 3 and not 4, stck = 2 1
slog "stck " %stck
     # prints the stack value and remove it, stck = 1
slog "stck " %stck
     # prints the stack value and remove it, stck =
slog "stck " %stck   # prints an empty value: the stack is empty

post %stck 1,2,3     # post: with or without commas
remove %stck 2       # remove a value from stack, stck = 3 1
remove %stck 10
     # does nothing: value not present into the stack
slog "stck " %stck
     # prints the stack value and remove it, stck = 1
slog "stck " %stck
     # prints the stack value and remove it, stck =
slog "stck " %stck   # prints an empty value: the stack is empty
exit                 # NB: always an exit at the end
```

**Output: luca-stack.out**

```
luca-stack: 24 steps compiled
luca-stack: stck 3
luca-stack: stck 2
luca-stack: stck 1
luca-stack: stck
luca-stack: stck 3
luca-stack: stck 2
luca-stack: stck 1
luca-stack: stck 3
luca-stack: stck 2
luca-stack: stck 1
luca-stack: stck
luca-stack: stck 3
luca-stack: stck 1
luca-stack: stck
exiting...
```

FIXME fifo bug FIXME

## Example 28: fifo

*Testing environment: Ccscript 2.5.1*

**Source: luca-fifo.scr**

```
fifo 3 %fifo        # fifo declaration
post %fifo 1         # insert a value, fifo = 1
post %fifo 2         # insert a value, fifo = 1 2
post %fifo 3         # insert a value, fifo = 1 2 3
slog "fifo " %fifo
     # prints the fifo value and remove it, fifo = 2 3
slog "fifo " %fifo
     # prints the fifo value and remove it, fifo = 3
slog "fifo " %fifo
     # prints the fifo value and remove it, fifo =
slog "fifo " %fifo
     # prints an empty value: the fifo is empty

post %fifo 1 2 3     # multiple post, fifo = 1 2 3
slog "fifo " %fifo
     # prints the fifo value and remove it, fifo = 2 3
slog "fifo " %fifo
     # prints the fifo value and remove it, fifo = 3
slog "fifo " %fifo
     # prints the fifo value and remove it, fifo =

post %fifo 1 2 3 4   # too many elements: the fifo has size 3
                     # the fourth value is ignored
slog "fifo " %fifo   # prints 1, fifo = 2 3
slog "fifo " %fifo
     # prints the fifo value and remove it, fifo = 3
slog "fifo " %fifo
     # prints the fifo value and remove it, fifo =
slog "fifo " %fifo   # prints an empty value: the fifo is empty

post %fifo 1,2,3     # post: with or without commas
remove %fifo 2       # remove a value from fifo, fifo = 1 3
remove %fifo 10
     # does nothing: value not present into the fifo
slog "fifo " %fifo
     # prints the fifo value and remove it, fifo = 3
slog "fifo " %fifo
     # prints the fifo value and remove it, fifo =
slog "fifo " %fifo   # prints an empty value: the fifo is empty
exit                 # NB: always an exit at the end
```

**Output: luca-fifo.out**

```
luca-fifo: 24 steps compiled
luca-fifo: fifo 1
luca-fifo: fifo 2
luca-fifo: fifo
luca-fifo: fifo
luca-fifo: fifo 1
```

```
luca-fifo: fifo 2
luca-fifo: fifo
luca-fifo: fifo 1
luca-fifo: fifo 2
luca-fifo: fifo
luca-fifo: fifo
luca-fifo: fifo 1
luca-fifo: fifo
luca-fifo: fifo
exiting...
```

The *sequence* structure is also dynamic, but doesn't remove a value after using it: when the last element is used, the next referenced is the first again. So from a not empty sequence is always possible getting values. With instructions *post* and *remove* is possible to add/remove values to/from sequence.

## Example 29: sequence

*Testing environment: Ccscript 2.5.1*

**Source: luca-sequence.scr**

```
sequence 3 %seq     # sequence declaration
post %seq 1         # insert a value, seq = 1
post %seq 2 3       # insert two values, seq = 1 2 3
slog "seq " %seq    # prints the sequence value 1
slog "seq " %seq    # prints the sequence value 2
slog "seq " %seq    # prints the sequence value 3
slog "seq " %seq    # prints the first value again: 1
remove %seq 2       # remove a value from sequence
remove %seq 10      # does nothing: value not present
                    # into the sequence
slog "seq " %seq    # prints the sequence value 3
slog "seq " %seq    # prints the sequence value 1
slog "seq " %seq    # prints the sequence value 3

sequence 3 %seq2    # another sequence
post %seq2 1 2 3 4  # too many elements: sequenze has size 3
                    # the fourth element is ignered
slog "seq2 " %seq2    # prints the sequence value 1
slog "seq2 " %seq2    # prints the sequence value 2
slog "seq2 " %seq2    # prints the sequence value 3
slog "seq2 " %seq2    # prints the sequence value 1 not 4

exit                # NB: always an exit at the end
```

**Output: luca-sequence.out**

```
luca-sequence: 19 steps compiled
luca-sequence: seq 1
luca-sequence: seq 2
luca-sequence: seq 3
luca-sequence: seq 1
```

```
luca-sequence: seq 3
luca-sequence: seq 1
luca-sequence: seq 3
luca-sequence: seq2 1
luca-sequence: seq2 2
luca-sequence: seq2 3
luca-sequence: seq2 1
exiting...
```

## 5.4 Array

Ccscript array are not dynamic, but have a finite and fixed size. Arrays support three different notation (with array %vett):

- Direct access: the $n^{th}$ element is referred by %vett.n;
- Index access: the $n$ $th$ element is refereed directly by %vett, but there is the need to set index %n.index to value n;
- Hash access: the $n^{th}$ element is refereed by %vett#n.

### Example 30: array

*Testing environment: Ccscript 2.5.1*

**Source: luca-array.scr**

```
array 3 %vett        # array declaration
set %vett.1 10       # setting values
set %vett.2 20
set %vett.3 30

    # basic notation

slog "vett.1 " %vett.1
slog "vett.2 " %vett.2
slog "vett.3 " %vett.3

    # index notation: useful for loops

set %vett.index 1
slog "element with index " %vett.index ": "%vett
set %vett.index 2
slog "element with index " %vett.index ": "%vett
set %vett.index 3
slog "element with index " %vett.index ": "%vett

    # hash notation: still useful for loops

set %tmp 1
slog "vett#%tmp: " %vett#%tmp
set %tmp 2
slog "vett#%tmp: " %vett#%tmp
```

```
set %tmp 3
slog "vett#%tmp: " %vett#%tmp
exit                # NB: always an exit at the end
```

**Output: luca-array.out**

```
luca-array: 20 steps compiled
luca-array: vett.1 10
luca-array: vett.2 20
luca-array: vett.3 30
luca-array: element with index 1: 10
luca-array: element with index 2: 20
luca-array: element with index 3: 30
luca-array: vett#%tmp: 10
luca-array: vett#%tmp: 20
luca-array: vett#%tmp: 30
exiting...
```

## 5.5 List

A list is a set of variables separated by a token (the default token is ',' and is stored in %script.token). To access the *nth* element there is the %list.n notation.

### Example 31: list, definition, access

*Testing environment: Ccscript 2.5.1*

**Source: luca-list.scr**

```
set %list1 "a,b,c,d,e"
slog "list1: " %list1
slog "list1 second element: " %list1.2  # direct access
set %list2 "f-g-h-i-l"
slog "list2: " %list2
slog "list2 second element (token ,): " %list2.2   # empty value
set %script.token "-"       # changes the token symbol
slog "list2 second element (token -): " %list2.2   # right value
exit                    # NB: always an exit at end
```

**Output: luca-list.out**

```
luca-list: 9 steps compiled
luca-list: list1: a,b,c,d,e
luca-list: list1 second element: b
luca-list: list2: f-g-h-i-l
luca-list: list2 second element (token ,):
luca-list: list2 second element (token -): g
exiting...
```

The fifth line has no value because the token '-' is unrecognized, in the sixth one the output is right because %script.token changed to '-'.

In the package string (*use string*) there are some useful functions for lists. To create a list from several variables there is the instruction *string.pack*, so in the following examples lists are also called packed strings.

## Example 32: packed string, definition

*Testing environment: Ccscript 2.5.1*

**Source: luca-pack1.scr**

```
use string
string.pack %pack1  1 2 3          # basic definition
slog %pack1
string.pack %pack2 token=- 1 2 3
     # use "-" instead of dafault = ","
slog %pack2
string.pack %pack3 size=3 1 2 3
     # with size=3 not enough space for
slog %pack3                        # all values
string.pack %pack4 prefix="start" suffix="end" 1 2 3
            # defines a starting and ending values
slog %pack4
string.pack %pack5 size=20 fill="FILL" 1 2 3
            # add as many values "FILL" as possible
slog %pack5
string.pack %pack6 size=22 fill="FILL" 1 2 3
            # add as many values "FILL" as possible
            # the last "FILL" value i trunced by size
slog %pack6

exit                    # NB: always an exit at end
```

**Output: luca-pack1.out**

```
luca-pack1: 14 steps compiled
luca-pack1: 1,2,3
luca-pack1: 1-2-3
luca-pack1: 1,2
luca-pack1: start,1,2,3,end
luca-pack1: 1,2,3,FILL,FILL,FILL
luca-pack1: 1,2,3,FILL,FILL,FILL,F
exiting...
```

In the output line there are the results of the different kind of list creation.

The *string.pack* instruction allows appending, while *string.repack* and *string.clear* allow list redefine and list clear respectively.

## Example 33: packed string, append, clear, redefine

*Testing environment: Ccscript 2.5.1*

**Source: luca-pack2.scr**

```
use string
string.pack %pack  1 2 3          # basic definition
slog %pack
string.pack %pack  4 5 6          # basic appending
slog %pack
string.repack token='-' %pack 7 8 9    # basic redefine
slog %pack
string.clear %pack                # clear the string
slog %pack
exit                    # NB: always an exit at end
```

**Output: luca-pack2.out**

```
luca-pack2: 10 steps compiled
luca-pack2: 1,2,3
luca-pack2: 1,2,3,4,5,6
luca-pack2: 7-8-9
luca-pack2:
exiting...
```

With *string.unpack* instruction is possible to extract values from a list, while with *string.cut* values are removed from list too.

## Example 34: packed string, get value, remove value

*Testing environment: Ccscript 2.5.1*

**Source: luca-pack3.scr**

```
use string
string.pack %pack1  10 20 30    # basic definition
string.pack token=- %pack2  40 50 60
slog %pack1 " " %pack2
       # gets 3 values from pack and places into variables
string.unpack %pack1 %v1 %v2 %v3
slog "v1:"%v1 " v2:" %v2 " v3:" %v3
       # gets value from pack starting by the second, the first
       # value is the number 1
string.unpack offset=2 %pack1 %v4 %v5
slog "v4:"%v4 " v5:" %v5
       # this doesn't work: the token is not the default
string.unpack %pack2 %v4 %v5 %v6
slog "v4: "%v4 " v5:" %v5 " v6:" %v6
       # this work: the token is defined
string.unpack token=- %pack2 %v1 %v2 %v3
slog "v1:"%v1 " v2:" %v2 " v3:" %v3
       # gets 3 values from pack, places into variables
       # and remove them
string.cut %pack1 %v1 %v2 %v3
       # now %pack1 is empty
slog "v1:"%v1 " v2:" %v2 " v3:" %v3 " pack1: " %pack1
```

```
       # get and remove the second value from %pack2
string.cut token=- offset=2 %pack2 %v5
slog "v5:"%v5 " pack2: " %pack2
exit                    # NB: always an exit at end
```

**Output: luca-pack3.out**

```
luca-pack3: 17 steps compiled
luca-pack3: 10,20,30 40-50-60
luca-pack3: v1:10 v2:20 v3:30
luca-pack3: v4:20 v5:30
luca-pack3: v4: 40-50-60 v5:30 v6:
luca-pack3: v1:40 v2:50 v3:60
luca-pack3: v1:30 v2:20 v3:10 pack1:
luca-pack3: v5:50 pack2: 40-60
exiting...
```

## 5.6 Duplication

The instruction *dup* allows the duplication of a data structure.

### Example 35: dup

*Testing environment: Ccscript 2.5.1*

**Source: luca-dup.scr**

```
sequence 5 %seq
post %seq  1 2 3 4     # five values in random order
dup %seq %seq2         # sequenze duplication
stack 5 %stck
post %stck 1 2 3 4     # five values in random order
dup %stck %stck2       # sequenze duplication
slog "original seq: " %seq " original stack: " %stck
slog "seq:" %seq " seq2:" %seq2 " stck:" %stck " stck2:" %stck2
slog "seq:" %seq " seq2:" %seq2 " stck:" %stck " stck2:" %stck2
slog "seq:" %seq " seq2:" %seq2 " stck:" %stck " stck2:" %stck2
slog "seq:" %seq " seq2:" %seq2 " stck:" %stck " stck2:" %stck2
exit                    # NB: always an exit at end
```

**Output: luca-dup.out**

```
luca-dup: 12 steps compiled
luca-dup: original seq: 1 original stack: 4
luca-dup: seq:2 seq2:1 stck:3 stck2:4
luca-dup: seq:3 seq2:2 stck:2 stck2:3
luca-dup: seq:4 seq2:3 stck:1 stck2:2
luca-dup: seq:1 seq2:4 stck: stck2:1
exiting...
```

# 6 Properties

A Ccscript property is basically a function applied to a variable using the dot notation *%var.property*; this property function can return an internal value, like the maximun size of dynamic structure, or something else. Many property are defined into external package, so there's the need to import them (*use package*).

## 6.1 Basic

The *type* and *size* property get the basic information about data structures or variables.

### Example 36: type, size

*Testing environment: Ccscript 2.5.1*

**Source: luca-typesize.scr**

```
use date
use time
use string
set %v1 10
set %v2 text
set.date %v3 20031225
set.time %v4 115700
slog "number v1,val: "%v1 " type: " %v1.type ", size:" %v1.size
slog "string v2,val: "%v2 " type: " %v2.type ", size:" %v2.size
slog "date v3,val: "%v3 " type: " %v3.type ", size:" %v3.size
slog "time v4,val: "%v4 " type: " %v4.type ", size:" %v4.size

sequence 3 %v5
fifo 4 %v6
stack 5 %v7
array 6 %v8
string.pack %v9 1 2 3
counter %v10
slog "sequence v5 type: " %v5.type ", size: " %v5.size
slog "fifo v6 type: " %v6.type ", size: " %v6.size
slog "stack v7 type: " %v7.type ", size: " %v7.size
slog "array v8 type: " %v8.type ", size: " %v8.size
slog "pack v9 type: " %v9.type ", size: " %v9.size
slog "counter v10 type: " %v10.type " size: " %v10.size
exit                    # NB: always an exit at end
```

**Output: luca-typesize.out**

```
luca-typesize: 24 steps compiled
luca-typesize: number v1,val: 10 type: string, size:64
luca-typesize: string v2,val: text type: string, size:64
luca-typesize: date v3,val: 12/25/2003 type: string, size:10
luca-typesize: time v4,val: 11:57:00 type: string, size:8
luca-typesize: sequence v5 type: sequence, size: 62
luca-typesize: fifo v6 type: fifo, size: 61
luca-typesize: stack v7 type: stack, size: 60
```

```
luca-typesize: array v8 type:
luca-typesize: pack v9 type: string, size: 64
luca-typesize: counter v10 type: counter size: 11
exiting...
```

## 6.2 Date

In the *date* package there are some useful properties to get year, month, day values or day name.
FIXME (monthof) FIXME

### Example 37: date

*Testing environment: Ccscript 2.5.1*

**Source: luca-dateProp.scr**

```
use date
set.date %dayDate "20031225"           # Chrismas 2003
slog "dayDate: " %dayDate
slog %dayDate.date " " %dayDate.monthof " " %dayDate.weekday
slog %dayDate.year " " %dayDate.month " " %dayDate.day
set %dayStr "20031225"           # Chrismas 2003
slog "dayStr: " %dayStr
slog %dayStr.date " " %dayStr.monthof " " %dayStr.weekday
slog %dayStr.year " " %dayStr.month " " %dayStr.day
exit                       # NB: always an exit at the end
```

**Output: luca-dateProp.out**

```
luca-dateProp: 12 steps compiled
luca-dateProp: dayDate 12/25/2003
luca-dateProp: string 10 10
luca-dateProp: 20031225 20031225 thursday
luca-dateProp: 2003 12 25
luca-dateProp: dayStr 20031225
luca-dateProp: string 8 64
luca-dateProp: 20031225 20031225 thursday
luca-dateProp: 2003 12 25
exiting...
```

## 6.3 Time

In the *time* package there are some useful properties to get hour, minute, second values.

### Example 38: time

*Testing environment: Ccscript 2.5.1*

**Source: luca-timeProp.scr**

```
use time
set.time %nowTime "115740"           # a time
slog "nowTime " %nowTime
slog %nowTime.time
slog %nowTime.hour " " %nowTime.minute " " %nowTime.second
set %nowStr "125740"           # a time
slog "nowStr " %nowStr
slog %nowStr.hour " " %nowStr.minute " " %nowStr.second
exit                       # NB: always an exit at the end
```

**Output: luca-timeProp.out**

```
luca-timeProp: 9 steps compiled
luca-timeProp: nowTime 11:57:40
luca-timeProp: 115740
luca-timeProp: 11 57 40
luca-timeProp: nowStr 125740
luca-timeProp: 12 57 40
exiting...
```

## 6.4 String

In the *string* package there are some useful properties to uppercase, lowercase and capitalize a string.

## Example 39: string

*Testing environment: Ccscript 2.5.1*

**Source: luca-stringProp.scr**

```
use string
set %var "abcd"           # a generic variable
slog "uppecase: " %var.upper
set %var "ABCD"
slog "lowercase: " %var.lower
set %var "abcd"
slog "capitalize: " %var.capitalize
set %var "  abcd  "       # with spaces
slog "with spaces:-" %var "-"
slog "without spaces:-" %var.trim "-"
exit                       # NB: always an exit at the end
```

**Output: luca-stringProp.out**

```
luca-stringProp: 11 steps compiled
luca-stringProp: uppecase: ABCD
luca-stringProp: lowercase: abcd
luca-stringProp: capitalize: Abcd
luca-stringProp: with spaces:-  abcd  -
luca-stringProp: without spaces:-abcd-
exiting...
```

### 6.5 Miscellaneous

The property *each* of package *digits* creates a list of all digits of a string.

### Example 40: each

*Testing environment: Ccscript 2.5.1*

**Source: luca-each.scr**

```
use digits
set %str "ccscript"
set %list %str.each
slog "list of digits: " %list
exit                    # NB: always an exit at end
```

**Output: luca-each.out**

```
luca-each: 5 steps compiled
luca-each: list of digits: c,c,s,c,r,i,p,t
exiting...
```

## 7 Conditional structures, loops

In this section there are examples for basic loops, conditional structures and conditional loops.

### 7.1 Finite loops

To loop $n$ times there are the following instructions:

- *repeat*: to loop $n$ times without an index;
- *for*: to loop with the indicated index values;
- *foreach*: to loop with index values from a list structure.

these instructions are all closed by the instruction *loop*.

### Example 41: repeat

*Testing environment: Ccscript*

**Source: luca-repeat.scr**

```
counter %count
repeat 3            # loop without index or condition
  slog "counter: " %count
loop                # closes the loop
exit                # NB: always an exit at the end
```

**Output: luca-repeat.out**

```
luca-repeat: 5 steps compiled
luca-repeat: counter: 1
luca-repeat: counter: 2
luca-repeat: counter: 3
exiting...
```

## Example 42: for

*Testing environment: Ccscript*

**Source: luca-for.scr**

```
for %index 1 2 3 4 a b string       # any value
  slog "index: " %index
loop                # closes the loop
exit                # NB: always an exit at the end
```

**Output: luca-for.out**

```
luca-for: 4 steps compiled
luca-for: index: 1
luca-for: index: 2
luca-for: index: 3
luca-for: index: 4
luca-for: index: a
luca-for: index: b
luca-for: index: string
exiting...
```

## Example 43: foreach

*Testing environment: Ccscript 2.5.1*

**Source: luca-foreach.scr**

```
use string   # to use packed string
foreach %index "1,2,3,4,a,b,string"       # any value
  slog "index: " %index
loop                # closes the loop

string.pack %pack 5 6 7 c d "any value"
slog "with packed string: " %pack
foreach %index %pack
  slog "index: " %index
loop                # closes the loop
exit                # NB: always an exit at the end
```

**Output: luca-foreach.out**

```
luca-foreach: 10 steps compiled
luca-foreach: index: 1
luca-foreach: index: 2
luca-foreach: index: 3
luca-foreach: index: 4
luca-foreach: index: a
luca-foreach: index: b
luca-foreach: index: string
luca-foreach: with packed string: 5,6,7,c,d,any value
luca-foreach: index: 5
luca-foreach: index: 6
luca-foreach: index: 7
luca-foreach: index: c
luca-foreach: index: d
luca-foreach: index: any value
exiting...
```

## 7.2 Conditions

Ccscript conditions are of two types: numerical ones and string ones.

The numerical conditions are:

- -eq or =: equal;
- -ne or <>: not equal;
- -lt or <: less then;
- -le or <=: less equal;
- -ge or >=: greater equal;
- -gt or >: greater then.

The next two examples have the same output.

### Example 44: numerical conditions

*Testing environment: Ccscript 2.5.1*

**Source: luca-cond1.scr**

```
for %index 1 2 3 4        # any value
  slog "index " %index
  if %index -eq 1 then slog "index equals 1"
  if %index -ne 1 then slog "index not equals 1"
  if %index -lt 3 then slog "index less than 3"
  if %index -le 3 then slog "index less equal 3"
  if %index -ge 2 then slog "index greater equal 2"
  if %index -gt 2 then slog "index greater than 2"
loop                # closes the loop
exit                # NB: always an exit at the end
```

**Output: luca-cond1.out**

```
luca-cond1: 16 steps compiled
luca-cond1: index 1
luca-cond1: index equals 1
luca-cond1: index less than 3
luca-cond1: index less equal 3
luca-cond1: index 2
luca-cond1: index not equals 1
luca-cond1: index less than 3
luca-cond1: index less equal 3
luca-cond1: index greater equal 2
luca-cond1: index 3
luca-cond1: index not equals 1
luca-cond1: index less equal 3
luca-cond1: index greater equal 2
luca-cond1: index greater than 2
luca-cond1: index 4
luca-cond1: index not equals 1
luca-cond1: index greater equal 2
luca-cond1: index greater than 2
exiting...
```

## Example 45: numerical conditions, symbols

*Testing environment: Ccscript 2.5.1*

**Source: luca-cond2.scr**

```
for %index 1 2 3 4        # any value
  slog "index " %index
  if %index = 1 then slog "index equals 1"
  if %index <> 1 then slog "index not equals 1"
  if %index < 3 then slog "index less than 3"
  if %index <= 3 then slog "index less equal 3"
  if %index >= 2 then slog "index greater equal 2"
  if %index > 2 then slog "index greater than 2"
loop                  # closes the loop
exit                  # NB: always an exit at the end
```

**Output: luca-cond2.out**

```
luca-cond2: 16 steps compiled
luca-cond2: index 1
luca-cond2: index equals 1
luca-cond2: index less than 3
luca-cond2: index less equal 3
luca-cond2: index 2
luca-cond2: index not equals 1
luca-cond2: index less than 3
luca-cond2: index less equal 3
luca-cond2: index greater equal 2
luca-cond2: index 3
luca-cond2: index not equals 1
```

```
luca-cond2: index less equal 3
luca-cond2: index greater equal 2
luca-cond2: index greater than 2
luca-cond2: index 4
luca-cond2: index not equals 1
luca-cond2: index greater equal 2
luca-cond2: index greater than 2
exiting..
```

The string conditions are:

- .eq. or '==': equal;
- .ne. or '!=': not equal;
- $i< or '$+': returns true if the argument on the left is found in the argument on the right using strnicmp(3). If the argument on the left is shorter than the argument on the right, only the first n characters will be compared, where n is the length of the argument on the left. This operator is case-insensitive;
- $> or '$-': returns true if the argument on the left is found in the argument on the right using strnicmp(3). If the argument on the left is shorter than the argument on the right, then only the n rightmost characters will be compared, where n is the length of the argument on the left. This operator is case-insensitive.

The next two examples have the same output.

## Example 46: string conditions

*Testing environment: Ccscript 2.5.1*

**Source: luca-cond3.scr**

```
for %index a aa bb cc       # any value
  slog "index " %index
  if %index .eq. a then slog "index equals a"
  if %index .ne. a then slog "index not equals a"
  if %index $< aab then slog "index is a prefix of aab"
  if %index $> cbb then slog "index is a suffix of cbb"
loop                 # closes the loop
exit                 # NB: always an exit at the end
```

**Output: luca-cond3.out**

```
luca-cond3: 12 steps compiled
luca-cond3: index a
luca-cond3: index equals a
luca-cond3: index is a prefix of aab
luca-cond3: index aa
luca-cond3: index not equals a
luca-cond3: index is a prefix of aab
luca-cond3: index bb
luca-cond3: index not equals a
luca-cond3: index is a suffix of cbb
luca-cond3: index cc
```

```
luca-cond3: index not equals a
exiting...
```

## Example 47: string conditions, symbols

*Testing environment: Ccscript 2.5.1*

**Source: luca-cond4.scr**

```
for %index a aa bb cc       # any value
  slog "index " %index
  if %index == a then slog "index equals a"
  if %index != a then slog "index not equals a"
  if %index $+ aab then slog "index is a prefix of aab"
  if %index $- cbb then slog "index is a suffix of cbb"
loop                 # closes the loop
exit                 # NB: always an exit at the end
```

**Output: luca-cond4.out**

```
luca-cond4: 12 steps compiled
luca-cond4: index a
luca-cond4: index equals a
luca-cond4: index not equals a
luca-cond4: index is a prefix of aab
luca-cond4: index aa
luca-cond4: index not equals a
luca-cond4: index is a prefix of aab
luca-cond4: index bb
luca-cond4: index not equals a
luca-cond4: index is a suffix of cbb
luca-cond4: index cc
luca-cond4: index not equals a
exiting...
```

For strings Ccscript allows a case sensitive condition:

- $: returns true if the string given by the left argument is found in the string given by the right argument using strstr(2); this operator is case-sensitive.

## Example 48: string conditions, case sensitive

*Testing environment: Ccscript 2.5.1*

**Source: luca-cond5.scr**

```
for %index aa AA "aa bb" "bb aa" "AA bb" "b aa b" "b aaa"
  slog "index " %index
  if aa $ %index then slog "index contains aa"
loop                 # closes the loop
exit                 # NB: always an exit at the end
```

**Output: luca-cond5.out**

```
luca-cond5: 6 steps compiled
luca-cond5: index aa
luca-cond5: index contains aa
luca-cond5: index AA
luca-cond5: index aa bb
luca-cond5: index contains aa
luca-cond5: index bb aa
luca-cond5: index contains aa
luca-cond5: index AA bb
luca-cond5: index b aa b
luca-cond5: index contains aa
luca-cond5: index b aaa
luca-cond5: index contains aa
exiting...
```

Two or more conditions can be composed using the operators *and* and *or*.

### Example 49: conditions, and, or

*Testing environment: Ccscript 2.5.1*

**Source: luca-cond6.scr**

```
for %index 1 2 3 4        # any value
  slog "index " %index
  if %index -eq 1 or %index -eq 3 then slog "index is odd"
  if %index -gt 1 and %index -lt 4 then slog "index is 2 or 3"
loop                 # closes the loop
exit                 # NB: always an exit at the end
```

**Output: luca-cond6.out**

```
luca-cond6: 8 steps compiled
luca-cond6: index 1
luca-cond6: index is odd
luca-cond6: index 2
luca-cond6: index is 2 or 3
luca-cond6: index 3
luca-cond6: index is odd
luca-cond6: index is 2 or 3
luca-cond6: index 4
```

## 7.3 Conditional structures

Ccscript has two conditional structures:

- if
  *if* condition *then*
  *if* condition *then endif*
  *if* condition *then else endif*
- case
  *case* condition *endcase*
  *case* condition *otherwise endcase*

## Example 50: if

*Testing environment: Ccscript 2.5.1*

**Source: luca-if.scr**

```
for %index 1 2 3 4        # any value
  if %index -eq 4 then slog "index is 4"
      # single line statement
  if %index -eq 1 or %index -eq 3
  then         # in this block as many statement as we wish
    slog "index " %index
    slog "index is odd"
  else
    slog "index " %index
    slog "index is even"
  endif
loop                # closes the loop
exit                # NB: always an exit at the end
```

**Output: luca-if.out**

```
luca-if: 13 steps compiled
luca-if: index 1
luca-if: index is odd
luca-if: index 2
luca-if: index is even
luca-if: index 3
luca-if: index is odd
luca-if: index is 4
luca-if: index 4
luca-if: index is even
exiting....
```

In the following example there is a *case* with mutual exclusive condition:

## Example 51: case, mutual exclusive conditions

*Testing environment: Ccscript 2.5.1*

**Source: luca-case1.scr**

```
for %index 0 1 2 3 4        # any value
  case %index -eq 1 or %index -eq 3
    slog "index " %index
    slog "index is odd"
  case %index -eq 2 or %index -eq 4
    slog "index " %index
    slog "index is even"
  otherwise
    slog "index " %index " is null"
  endcase
loop                # closes the loop
exit                # NB: always an exit at the end
```

**Output: luca-case1.out**

```
e1: 12 steps compiled
luca-case1: index 0 is null
luca-case1: index 1
luca-case1: index is odd
luca-case1: index 2
luca-case1: index is even
luca-case1: index 3
luca-case1: index is odd
luca-case1: index 4
luca-case1: index is even
exiting...
```

In the following example there is a *case* without mutual exclusive condition:

## Example 52: case, overlap conditions

*Testing environment: Ccscript 2.5.1*

**Source: luca-case2.scr**

```
for %index 0 1 2 3 4        # any value
  slog "index " %index
  case %index -eq 1 or %index -eq 4
    slog "index is a square"
  case %index -eq 1 or %index -eq 3
    slog "index is odd"
  case %index -eq 2 or %index -eq 4
    slog "index is even"
  endcase
loop                # closes the loop
exit                # NB: always an exit at the end
```

**Output: luca-case2.out**

```
luca-case2: 11 steps compiled
luca-case2: index 0
luca-case2: index 1
luca-case2: index is a square
luca-case2: index 2
luca-case2: index is even
luca-case2: index 3
luca-case2: index is odd
luca-case2: index 4
luca-case2: index is a square
exiting...
```

By output it's possible to see that in the *case* structure when one condition is true all other condition are ignored.

## 7.4 Conditional loops

To make loop with initial and/or end conditions Ccscript has *do loop* structure:

- *do*: loop start, *do condition* to start a loop with a initial condition;
- *loop*: loop end, *loop condition* to close a loop with an ending condition.

The *loop* instruction supports conditions with commands *for*, *foreach* and *repear*, but in these cases is often used without condition.

### Example 53: do condition loop

*Testing environment: Ccscript 2.5.1*

**Source: luca-doloop1.scr**

```
set %index 0
do %index -lt 4    # condition before loop
  slog "index: " %index
  inc %index
loop               # closes the loop

do %index -lt 4    # can be never executed
  slog "index: " %index
  inc %index
loop               # closes the loop
exit               # NB: always an exit at the end
```

**Output: luca-doloop1.out**

```
luca-doloop1: 10 steps compiled
luca-doloop1: index: 0
luca-doloop1: index: 1
luca-doloop1: index: 2
luca-doloop1: index: 3
exiting...
```

### Example 54: do loop condition

*Testing environment: Ccscript 2.5.1*

**Source: luca-doloop2.scr**

```
set %index 0
do                    # no condition before loop
  slog "index: " %index
  inc %index
loop %index -lt 4    # condition after loop

do                    # always executed at least once
  slog "index: " %index
  inc %index
loop %index -lt 4    # condition aftere loop
exit               # NB: always an exit at the end
```

**Output: luca-doloop2.out**

```
luca-doloop2: 10 steps compiled
luca-doloop2: index: 0
luca-doloop2: index: 1
luca-doloop2: index: 2
luca-doloop2: index: 3
luca-doloop2: index: 4
exiting...
```

### Example 55: do condition loop condition

*Testing environment: Ccscript 2.5.1*

**Source: luca-doloop3.scr**

```
set %index1 0
set %index2 10
do  %index1 -lt 5    # condition before loop
  slog "index1: " %index1 " index2: " %index2
  inc %index1
  dec %index2
loop %index2 -gt 5    # condition after loop

exit               # NB: always an exit at the end
```

**Output: luca-doloop3.out**

```
luca-doloop3: 8 steps compiled
luca-doloop3: index1: 0 index2: 10
luca-doloop3: index1: 1 index2: 9
luca-doloop3: index1: 2 index2: 8
luca-doloop3: index1: 3 index2: 7
luca-doloop3: index1: 4 index2: 6
exiting...
```

## 7.5 Loop override

Every Ccscript loop behavior can be forced with two instruction:

- *continue*: continue a loop beginning the next cycle;
- *break*: break a loop jumping to the next instruction after the loop.

Both instruction can have a condition.

### Example 56: break, continue

*Testing environment: Ccscript 2.5.1*

**Source: luca-breakcontinue.scr**

```
for %index 1 2 3 4 5   # any value
  slog "before break and continue index: " %index
  if %index -eq 2
  then
     slog "continue match"
     continue        # continue with no condition
  endif
  if %index -eq 3
  then
     slog "break match"
     break           # break with no condition
  endif
  slog "after break and continue index: " %index
loop                 # closes the loop
slog "Second loop"
for %index 1 2 3 4 5   # any value
  slog "before break and continue index: " %index
  continue %index -eq 2  # continue with a condition
  break %index -eq 3     # break with a condition
  slog "after break and continue index: " %index
loop                 # closes the loop
exit                 # NB: always an exit at the end
```

**Output: luca-breakcontinue.out**

```
luca-breakcontinue: 22 steps compiled
luca-breakcontinue: before break and continue index: 1
luca-breakcontinue: after break and continue index: 1
luca-breakcontinue: before break and continue index: 2
luca-breakcontinue: continue match
luca-breakcontinue: before break and continue index: 3
luca-breakcontinue: break match
luca-breakcontinue: Second loop
luca-breakcontinue: before break and continue index: 1
luca-breakcontinue: after break and continue index: 1
luca-breakcontinue: before break and continue index: 2
luca-breakcontinue: before break and continue index: 3
exiting...
```

From output it's important to note that:

- in the line 5 the loop continues: there is no line "after continue and break index: 2", but "before continue and break index: 3";
- in the line 7 the loop breaks: there is no line "after continue e break index 3", but "Second loop";
- in the line 11 the loop continues: there is no line "after continue and break index: 2", but "before continue and break index: 3";
- in the line 12 the loop breaks: there is no line "after continue and break index: 2", but exit.

# 8 Labels and jumps

Ccscript supports two kind of labels, one defined with *label* instruction, the other defined with *::label* syntax.

## 8.1 Skip

Label defined with *label* instruction can be used with *skip* instruction.

### Example 57: label, skip

*Testing environment: Ccscript 2.5.1*

**Source: luca-labelskip.scr**

```
set %index 0
label lab1          # label definition
slog "after lab1"
label lab2          # label definition
slog "after lab2"
label lab3          # label definition
slog "after lab3"
slog ""             # empty line
inc %index          # counts the loop
if %index -eq 1 then skip lab1
if %index -eq 2 then skip lab2
if %index -eq 3 then skip lab3
exit                # NB: always an exit at the end
```

**Output: luca-labelskip.out**

```
luca-labelskip: 16 steps compiled
luca-labelskip: after lab1
luca-labelskip: after lab2
luca-labelskip: after lab3
luca-labelskip:
luca-labelskip: after lab1
luca-labelskip: after lab2
luca-labelskip: after lab3
luca-labelskip:
luca-labelskip: after lab2
luca-labelskip: after lab3
luca-labelskip:
luca-labelskip: after lab3
luca-labelskip:
exiting...
```

The output shows this kind of loop.

## 8.2 Goto

Label defined with *::label* syntax can be used with *goto* and *gosub* instructions.

## Example 58: label, goto

*Testing environment: Ccscript 2.5.1*

**Source: luca-labelgoto.scr**

```
  set %index 0
  goto ::lab1
::lab1        # label definition
  slog "after lab1"
  goto ::lab2
::lab2        # label definition
  slog "after lab2"
  goto ::lab3
::lab3        # label definition
  slog "after lab3"
  goto ::common
::common
  slog ""
  inc %index
  if %index -eq 1 then goto ::lab1
  if %index -eq 2 then goto ::lab2
  if %index -eq 3 then goto ::lab3
exit           # NB: always an exit at the end
```

**Output: luca-labelgoto.out**

```
luca-labelgoto: 2 steps compiled
luca-labelgoto::lab1: 2 steps compiled
luca-labelgoto::lab2: 2 steps compiled
luca-labelgoto::lab3: 2 steps compiled
luca-labelgoto::common: 9 steps compiled
luca-labelgoto::lab1: after lab1
luca-labelgoto::lab2: after lab2
luca-labelgoto::lab3: after lab3
luca-labelgoto::common:
luca-labelgoto::lab1: after lab1
luca-labelgoto::lab2: after lab2
luca-labelgoto::lab3: after lab3
luca-labelgoto::common:
luca-labelgoto::lab2: after lab2
luca-labelgoto::lab3: after lab3
luca-labelgoto::common:
luca-labelgoto::lab3: after lab3
luca-labelgoto::common:
exiting...
```

The output shows again a kind of loop.

The *goto* instruction supports variable assignment too.

## Example 59: label, goto/2

*Testing environment: Ccscript 2.5.1*

**Source: luca-labelgoto2.scr**

```
  set %index 0
  set %string first
  goto ::lab1
::lab1        # label definition
  slog "after lab1"
  goto ::lab2
::lab2        # label definition
  slog "after lab2"
  goto ::lab3
::lab3        # label definition
  slog "after lab3"
  goto ::common
::common
  slog "string: " %string
  inc %index
  if %index -eq 1 then goto ::lab1 %string="second"
  if %index -eq 2 then goto ::lab2 %string="third"
  if %index -eq 3 then goto ::lab3 %string="fourth"
exit           # NB: always an exit at the end
```

**Output: luca-labelgoto2.out**

```
luca-labelgoto2: 3 steps compiled
luca-labelgoto2::lab1: 2 steps compiled
luca-labelgoto2::lab2: 2 steps compiled
luca-labelgoto2::lab3: 2 steps compiled
luca-labelgoto2::common: 9 steps compiled
luca-labelgoto2::lab1: after lab1
luca-labelgoto2::lab2: after lab2
luca-labelgoto2::lab3: after lab3
luca-labelgoto2::common: string: first
luca-labelgoto2::lab1: after lab1
luca-labelgoto2::lab2: after lab2
luca-labelgoto2::lab3: after lab3
luca-labelgoto2::common: string: second
luca-labelgoto2::lab2: after lab2
luca-labelgoto2::lab3: after lab3
luca-labelgoto2::common: string: third
luca-labelgoto2::lab3: after lab3
luca-labelgoto2::common: string: fourth
exiting...
```

The *if* structure support implicit *goto ::label* statement.

## Example 60: label, if

*Testing environment: Ccscript 2.5.1*

**Source: luca-labelif.scr**

```
  set %index 0
  goto ::lab1
::lab1        # label definition
```

```
  slog "after lab1"
  goto ::lab2
::lab2        # label definition
  slog "after lab2"
  goto ::lab3
::lab3        # label definition
  slog "after lab3"
  goto ::common
::common
  inc %index
  if %index -eq 1 ::lab1
  if %index -eq 2 ::lab2
  if %index -eq 3 ::lab3
exit           # NB: always an exit at the end
```

**Output: luca-labelif.out**

```
luca-labelif::lab1: 2 steps compiled
luca-labelif::lab2: 2 steps compiled
luca-labelif::lab3: 2 steps compiled
luca-labelif::common: 5 steps compiled
luca-labelif::lab1: after lab1
luca-labelif::lab2: after lab2
luca-labelif::lab3: after lab3
luca-labelif::lab1: after lab1
luca-labelif::lab2: after lab2
luca-labelif::lab3: after lab3
luca-labelif::lab2: after lab2
luca-labelif::lab3: after lab3
luca-labelif::lab3: after lab3
exiting...
```

From output it's possible to that there is a goto behavior without a got statement.

## 8.3 Subroutine

The structure *gosub ::label . . . return* supports subroutines, with parameter passing and supporting return values. The basic structure is the following (where only global params are used):

### Example 61: gosub, global params

*Testing environment: Ccscript 2.5.1*

**Source: luca-gosub.scr**

```
counter count        # counter is global
call ::routine
call ::routine
call ::routine
slog "local: " %local # empty value: it's local...
exit           # NB: always an exit at the end

::routine
```

```
  set %local %count    # local to subroutine
  slog "counter: " %local
  return
```

**Output: luca-gosub.out**

```
luca-gosub: 6 steps compiled
luca-gosub::routine: 3 steps compiled
luca-gosub::routine: counter: 1
luca-gosub::routine: counter: 2
luca-gosub::routine: counter: 3
luca-gosub: local:
exiting...
```

To use returning value in the *return* statement:

## Example 62: gosub, ret value

*Testing environment: Ccscript 2.5.1*

**Source: luca-gosub2.scr**

```
counter count        # counter is global
call ::routine
slog "ret value: " %ret " localvar: " %localvar
      # %localvar is empty
call ::routine
slog "ret value: " %ret " localvar: " %localvar
      # %localvar is empty
call ::routine
slog "ret value: " %ret " localvar: " %localvar
      # %localvar is empty
exit           # NB: always an exit at the end

::routine
  set %localvar %count    # local to subroutine
  return %ret=%localvar
```

**Output: luca-gosub2.out**

```
luca-gosub2: 8 steps compiled
luca-gosub2::routine: 2 steps compiled
luca-gosub2: ret value: 1 localvar:
luca-gosub2: ret value: 2 localvar:
luca-gosub2: ret value: 3 localvar:
exiting...
```

To use local params:

## Example 63: gosub, local params

*Testing environment: Ccscript 2.5.1*

**Source: luca-gosub3.scr**

```
call ::routine %par=1
slog "value of par:" %par  # empty value
call ::routine %par=2
slog "value of par:" %par  # empty value
call ::routine %par=3
slog "value of par:" %par  # empty value
exit           # NB: always an exit at the end

::routine
  set %local %par    # local to subroutine
  slog "local: " %local
  return
```

**Output: luca-gosub3.out**

```
luca-gosub3: 7 steps compiled
luca-gosub3::routine: 3 steps compiled
luca-gosub3::routine: local: 1
luca-gosub3: value of par:
luca-gosub3::routine: local: 2
luca-gosub3: value of par:
luca-gosub3::routine: local: 3
luca-gosub3: value of par:
exiting...
```

## 8.4 Scope

Using subroutine changes the scope of local variables:

### Example 64: local and global variables

*Testing environment: Ccscript 2.5.1*

**Source: luca-scope.scr**

```
set %global "global from main"
gosub ::routine
slog "global     : " %global
slog "local.var  : " %localvar     # empty
slog "global.var : " %global.var   # with a value
exit           # NB: always an exit at the end

::routine
                # local scope variable
  set %localvar "local variable from routine"
                # global scope variable
  set %global.var "global variable from routine"
  slog "global     : " %global
  slog "local.var  : " %localvar
  slog "global.var : " %global.var
  return
```

**Output: luca-scope.out**

```
luca-scope: 6 steps compiled
luca-scope::routine: 6 steps compiled
luca-scope::routine: global     : global from main
luca-scope::routine: local.var  : local variable from routine
luca-scope::routine: global.var : global variable from routine
luca-scope: global     : global from main
luca-scope: local.var  :
luca-scope: global.var : global variable from routine
exiting...
```

Variable defined within a subroutine is in local scope, so out of the subroutine scope the variable is undefined.

# 9 Event

Ccscript is event driven scripting language. An event is something that happens out of the normal processing flow. When an event happens, the corresponding event handler, if present, is called.

## 9.1 Hangup

Hangup is one of the most important Bayonne's event, it notifies when a phone call ends by user.

### Example 65: Hangup

*Testing environment: Ccscript 2.5.1, Bayonne 1.0.10, Bayonne 1.2.4*

**Source: luca-hangup.scr**

```
repeat 3
  slog "pause 5 second: hangup to exit"
  sleep 5
loop                # closes the loop
slog "after loop: exit"
exit                # NB: always an exit at the end

^hangup
   slog "hangup event: exit"
   exit
```

**Output: luca-hangup.out**

```
without giving a Hangup

dummy(0): luca-hangup: pause 5 second: hangup to exit
dummy(0): luca-hangup: pause 5 second: hangup to exit
dummy(0): luca-hangup: pause 5 second: hangup to exit
dummy(0): luca-hangup: after loop: exit

giving a HangUp
```

```
dummy(0): luca-hangup: pause 5 second: hangup to exit
dummy(0): luca-hangup: pause 5 second: hangup to exit
dummy0: hangup...
dummy(0): luca-hangup: hangup event: exit
```

With *Dummy driver* to generate a hangup event is necessary to hit 'H' key.

## 9.2 Input

Every key pressed on the phone keyboard generates two events: a general one (*dtmf*) and a specific one (*0...9, star, pound*).

The *dtmf* event notifies when a key is pressed (every key from phone keyboard):

### Example 66: input dtmf

*Testing environment: Ccscript 2.5.1, Bayonne 1.0.10, Bayonne 1.2.4*

**Source: luca-input-dtmf.scr**

```
do
  sleep 5
loop                # closes the loop
slog "after loop: exit"
exit                # NB: always an exit at the end

^dtmf
   slog "dtmf received"
   goto luca-input-dtmf   # loops to start
^hangup
   slog "hangup event: exit"
   exit
```

**Output: luca-input-dtmf.out**

```
dummy0: digit 1...
dummy(0): luca-input-dtmf: dtmf received
dummy0: digit 2...
dummy(0): luca-input-dtmf: dtmf received
dummy0: digit 3...
dummy(0): luca-input-dtmf: dtmf received
dummy0: digit 4...
dummy(0): luca-input-dtmf: dtmf received
dummy0: digit 5...
dummy(0): luca-input-dtmf: dtmf received
dummy0: digit 6...
dummy(0): luca-input-dtmf: dtmf received
dummy0: digit 7...
dummy(0): luca-input-dtmf: dtmf received
dummy0: digit 8...
dummy(0): luca-input-dtmf: dtmf received
dummy0: digit 9...
dummy(0): luca-input-dtmf: dtmf received
```

```
dummy0: digit 0...
dummy(0): luca-input-dtmf: dtmf received
dummy0: digit #...
dummy(0): luca-input-dtmf: dtmf received
dummy0: digit *...
dummy(0): luca-input-dtmf: dtmf received
dummy0: hangup...
dummy(0): luca-input-dtmf: hangup event: exit
```

To get this output press all the phone keys, then hangup.

To notify for specific input tone there are events *0...9, pound, star* (for input 0...9, '#', '*' respectively).

## Example 67: input single tone

*Testing environment: Ccscript 2.5.1, Bayonne 1.0.10, Bayonne 1.2.4*

**Source: luca-input-single.scr**

```
do
  sleep 5
loop                # closes the loop
slog "after loop: exit"
exit                # NB: always an exit at the end

^0
   slog "tone 0 received"
   goto luca-input-single   # loops to start
^1
   slog "tone 1 received"
   goto luca-input-single   # loops to start
^2
   slog "tone 2 received"
   goto luca-input-single   # loops to start
^3
   slog "tone 3 received"
   goto luca-input-single   # loops to start
^4
   slog "tone 4 received"
   goto luca-input-single   # loops to start
^5
   slog "tone 5 received"
   goto luca-input-single   # loops to start
^6
   slog "tone 6 received"
   goto luca-input-single   # loops to start
^7
   slog "tone 7 received"
   goto luca-input-single   # loops to start
^8
   slog "tone 8 received"
   goto luca-input-single   # loops to start
^9
```

```
   slog "tone 9 received"
   goto luca-input-single   # loops to start
^pound
   slog "tone # received"
   goto luca-input-single   # loops to start
^star
   slog "tone * received"
   goto luca-input-single   # loops to start
^hangup
   slog "hangup event: exit"
   exit
```

**Output: luca-input-single.out**

```
dummy0: digit 1...
dummy(0): luca-input-single: tone 1 received
dummy0: digit 2...
dummy(0): luca-input-single: tone 2 received
dummy0: digit 3...
dummy(0): luca-input-single: tone 3 received
dummy0: digit 4...
dummy(0): luca-input-single: tone 4 received
dummy0: digit 5...
dummy(0): luca-input-single: tone 5 received
dummy0: digit 6...
dummy(0): luca-input-single: tone 6 received
dummy0: digit 7...
dummy(0): luca-input-single: tone 7 received
dummy0: digit 8...
dummy(0): luca-input-single: tone 8 received
dummy0: digit 9...
dummy(0): luca-input-single: tone 9 received
dummy0: digit 0...
dummy(0): luca-input-single: tone 0 received
dummy0: digit *...
dummy(0): luca-input-single: tone * received
dummy0: digit #...
dummy(0): luca-input-single: tone # received
dummy0: hangup...
dummy(0): luca-input-single: hangup event: exit
```

To get this output press all the phone keys, then hangup.

## 9.3 Scope

Event handler have specific scope, so using labels like *::label* implies event handler redefinition.

### Example 68: event scope

*Testing environment: Ccscript 2.5.1, Bayonne 1.0.10, Bayonne 1.2.4*

**Source: luca-eventScope.scr**

```
repeat 3
  slog "pause 3 second: press 1 or hangup"
  sleep 3
loop                 # closes the loop
slog "after first loop"
goto ::loop2

^1
   slog "tone 1 received"
   goto ::loop2
^hangup
   slog "hangup event: exit"
   exit

::loop2
repeat 3
  slog "pause 3 second: press 2 or hangup"
  sleep 3
loop                 # closes the loop
slog "after second loop"
exit                 # NB: always an exit at the end

^2
   slog "tone 2 received"
   exit
^hangup
   slog "hangup event: exit"
   exit
```

**Output: luca-eventScope.out**

```
dummy(0): luca-eventScope: pause 3 second: press 1 or hangup
dummy0: digit 2...
dummy(0): luca-eventScope: pause 3 second: press 1 or hangup
dummy0: digit 1...
dummy(0): luca-eventScope: tone 1 received
dummy(0): luca-eventScope::loop2: pause 3 second: press 2
          or hangup
dummy0: digit 1...
dummy(0): luca-eventScope::loop2: pause 3 second: press 2
          or hangup
dummy0: digit 2...
dummy(0): luca-eventScope::loop2: tone 2 received
```

In this example output is important to note that:

- the key 2 in the second line is ignored (no handler `^2` or `^dtmf` are present);
- the key 1 in the fourth line is a valid input (the handler `^1` is present);
- the key 1 in the seventh line is not more a valid input (the handler `^1` is not present);
- the key 2 in the ninth line is now valid (the handler `^2` is present).

# 10 Audio

To perform telephone output Bayonne uses the following instructions:

- play: to reproduce audio file;
- speak: to reproduce audio file composing phrases;
- tone: to generate beeps or other signals.

Audio files must have a particular format and characteristics. Supported formats are '.au' and '.wav'; the audio file must be 8 bit, 8 Khz, mono (these characteristics are due to phone standards).

## 10.1 Play

The *play* command doesn't need the extension of the file to play, it's determined from the default in the configuration file. The path used by Bayonne to find the audio file changes a lot by syntax used in the play command; often this path change with the values of global variable %session.voice. Assuming that %session.voice has VOICE value, there is (assuming '.au' the default extension):

- Bayonne 1.0.x:
  - `play audio`: Bayonne plays /usr/share/aapromps/VOICE/audio.au;
  - `play dir:audio`: Bayonne plays /usr/share/aapromps/dir/audio.au;
  - `play dir::audio`: Bayonne plays /home/bayonne/apps/dir/VOICE/-audio.au;
  - `play prefix=/dir audio`: Bayonne plays /dir/audio.au;
- Bayonne 1.2.x:
  - `play audio`: Bayonne plays /usr/local/share/bayonne/VOICE/audio-.au;
  - `play dir:audio`: Bayonne plays /usr/local/share/bayonne/sys/dir/audio.au;
  - `play dir::audio`: Bayonne plays /home/bayonne/dir/VOICE/audio-.au;
  - `play prefix=/dir audio`: Bayonne plays /dir/audio.au;

The *play* command supports more file to play.

For Bayonne 1.0.x:

### Example 69: play /1.0.x

*Testing environment: Ccscript 2.5.1, Bayonne 1.0.10*

**Source: luca-play1.scr**

```
set %session.voice VOICE
play audio         # use the default extension
play audio.au      # specific the extension
play audio.wav     # not the default extension
play dir:audio     # path changes
play dir::audio
play prefix=/dir audio
exit                    # NB: always an exit at end
```

**Output: luca-play1.out**

```
/usr/share/aaprompts/VOICE/audio.au: cannot open
/usr/share/aaprompts/VOICE/audio.au: cannot open
/usr/share/aaprompts/VOICE/audio.wav: cannot open
/usr/share/aaprompts/VOICE/dir:audio.au: cannot open
/home/bayonne/apps/dir/VOICE/audio.au: cannot open
/dir/audio: cannot open
```

For bayonne 1.2.x:

### Example 70: play /1.2.x

*Testing environment: Ccscript 2.5.1, Bayonne 1.2.4*

**Source: luca-play2.scr**

```
set %session.voice VOICE
play audio         # use the default extension
play audio.au      # specific the extension
play audio.wav     # not the default extension
play dir:audio     # path changes
play dir::audio
play prefix=/dir audio
exit                    # NB: always an exit at end
```

**Output: luca-play2.out**

```
/usr/local/share/bayonne/VOICE/audio.au: cannot open
/usr/local/share/bayonne/VOICE/audio.au: cannot open
/usr/local/share/bayonne/VOICE/audio.wav: cannot open
/usr/local/share/bayonne/sys/dir/audio.au: cannot open
/home/bayonne/dir/VOICE/audio.au: cannot open
/dir/audio: cannot open
```

## 10.2 Speak

The *speak* command can play audio file like *play*:

For Bayonne 1.0.x:

### Example 71: speak /1.0.x

*Testing environment: Ccscript 2.5.1, Bayonne 1.0.10*

**Source: luca-speak1.scr**

```
set %session.voice VOICE
speak audio         # use the default extension
speak audio.au      # specific the extension
speak audio.wav     # not the default extension
speak dir:audio     # path changes
speak dir::audio
speak prefix=/dir audio
exit                    # NB: always an exit at end
```

**Output: luca-speak1.out**

```
/usr/share/aaprompts/VOICE/audio.au: cannot open
/usr/share/aaprompts/VOICE/audio.au: cannot open
/usr/share/aaprompts/VOICE/audio.wav: cannot open
/usr/share/aaprompts/VOICE/dir:audio.au: cannot open
/home/bayonne/apps/dir/VOICE/audio.au: cannot open
/usr/share/aaprompts/VOICE/audio.au: cannot open
```

For bayonne 1.2.x:

## Example 72: speak /1.2.x

*Testing environment: Ccscript 2.5.1, Bayonne 1.2.4*

**Source: luca-speak3.scr**

```
set %session.voice VOICE
speak audio         # use the default extension
speak audio.au      # specific the extension
speak audio.wav     # not the default extension
speak dir:audio     # path changes
speak dir::audio
speak prefix=/dir audio
exit                    # NB: always an exit at end
```

**Output: luca-speak3.out**

```
/usr/local/share/bayonne/VOICE/audio.au: cannot open
/usr/local/share/bayonne/VOICE/audio.au: cannot open
/usr/local/share/bayonne/VOICE/audio.wav: cannot open
/usr/local/share/bayonne/sys/dir/audio.au: cannot open
/home/bayonne/dir/VOICE/audio.au: cannot open
/usr/local/share/bayonne/VOICE/audio.au: cannot open
```

But doesn't support 'prefix' directive (as you can see from the last output line).
To play audio file *play* is better than *speak*, The *speak* command is used to compose vocal phrases.
The *speak* supports a lot of directives (&directive):

- &spell: plays all digits of a string or of a number;
- &number: vocalize a number composing thousands, hundreds, tens, unit, ...

Digits, numbers, alphabetic letters used by *speak* are found (by default) in /usr/-share/aaprompts/ (with VOICE subdirectories).

For Bayonne 1.0.x:

## Example 73: speak, spell, number /1.0.x

*Testing environment: Ccscript 2.5.1, Bayonne 1.0.10*

**Source: luca-speak2.scr**

```
set %session.voice VOICE
set %var1 123
set %var2 "string"
slog  %var1 " " %var2
speak %var1
speak &number %var1
speak &spell %var1
speak %var2
speak &spell %var2
exit                    # NB: always an exit at end
```

**Output: luca-speak2.out**

```
dummy(0): luca-speak2: 123 string
/usr/share/aaprompts/VOICE/123.au: cannot open
/usr/share/aaprompts/VOICE/1.au: cannot open
/usr/share/aaprompts/VOICE/1.au: cannot open
/usr/share/aaprompts/VOICE/string.au: cannot open
/usr/share/aaprompts/VOICE/s.au: cannot open
```

For Bayonne 1.2.x:

## Example 74: speak, spell, number /1.2.x

*Testing environment: Ccscript 2.5.1, Bayonne 1.2.4*

**Source: luca-speak4.scr**

```
set %session.voice VOICE
set %var1 123
set %var2 "string"
slog  %var1 " " %var2
speak %var1
speak &number %var1
speak &spell %var1
speak %var2
speak &spell %var2
exit                    # NB: always an exit at end
```

**Output: luca-speak4.out**

```
/usr/local/share/bayonne/VOICE/123.au: cannot open
/usr/local/share/bayonne/VOICE/1.au: cannot open
/usr/local/share/bayonne/VOICE/1.au: cannot open
/usr/local/share/bayonne/VOICE/string.au: cannot open
/usr/local/share/bayonne/VOICE/s.au: cannot open
```

By the output it's possible to see that:

- speak &number %var1 say "1 hundred and twenty 3" (in the example audio file are not present, so speak stops after first file fail);
- speak &spell %var1 say "1, 2, 3" (in the example audio file are not present, so speak stops after first file fail);

- speak &spell %var2 say "s, t, r, i, n, g" (in the example audio file are not present, so speak stops after first file fail);

Using, for examples, VOICE = UsEngM it's possible to get correct audio output.

# 11 Input

Bayonne handles two types of input: audio/voice and keyboard digits. The first one is got with *record* command, the second one is got with *collect* command (keyboard digits can be read using events too).

## 11.1 Record

The *record* command receives audio/voice input and stores it in a file, the audio file created has the same type and format of audio files used by the *play* command. The *record* command has the format: `record audio length term` with:

- audio: name of audio file to record, the name can have extension (au/wav) or not (default is used);
- length: maximum record time, after this time the recording ends;
- term: digit set that can be used to end recording before timeout.

By default (of configuration file) recorded files are saved in */var/bayonne* for Bayonne 1.0.x ans into */var/lib/bayonne* for Bayonne 1.2.x. This path may be overridden via `record prefix=/path audio length term`.

For Bayonne 1.0.x:

### Example 75: record /1.0.x

*Testing environment: Ccscript 2.5.1, Bayonne 1.0.10*

**Source: luca-record.scr**

```
slog "before record: * to end"
record audio1 5 "*"          # /var/bayonne/audio1.au
slog "after first record"
slog "before record: # to end"
record prefix=/tmp audio2 5 "#"      # /tmp/audio2.au
slog "after second record"
exit                 # NB: always an exit at the end
```

**Output: luca-record.out**

```
dummy(0): luca-record: before record: * to end
dummy0: digit *...
dummy(0): luca-record: after first record
dummy(0): luca-record: before record: # to end
dummy0: digit #...
dummy(0): luca-record: after second record
```

For Bayonne 1.2.x:

## Example 76: record /1.2.x

*Testing environment: Ccscript 2.5.1, Bayonne 1.2.4*

**Source: luca-record1.scr**

```
slog "before record: * to end"
record audio1 5 "*"      # /var/lib/bayonne/audio1.au
slog "after first record"
slog "before record: # to end"
record prefix=/tmp audio2 5 "#"      # /tmp/audio2.au
slog "after second record"
exit                  # NB: always an exit at the end
```

**Output: luca-record1.out**

```
dummy(0): luca-record: before record: * to end
dummy0: digit *...
dummy(0): luca-record: after first record
dummy(0): luca-record: before record: # to end
dummy0: digit #...
dummy(0): luca-record: after second record
```

The two recordings end by pressing term keys. You can see the files were created in the appropriate directories.
Changing term keys and specifying audio format:

## Example 77: record, au/wav

*Testing environment: Ccscript 2.5.1, Bayonne 1.0.10, Bayonne 1.2.4*

**Source: luca-record2.scr**

```
slog "before record: 1 or 2 to end"
record audio1.au 5 "12"  # 1.0.x: /var/bayonne/audio1.au
                         # 1.2.x: /var/lib/bayonne/audio1.au
slog "after first record"
slog "before record: 3 or 4 to end"
record prefix=/tmp audio2.wav 5 "34"  # /tmp/audio2.wav
slog "after second record"
exit                  # NB: always an exit at the end
```

**Output: luca-record2.out**

```
dummy(0): luca-record2: before record: 1 or 2 to end
dummy0: digit 1...
dummy(0): luca-record2: after first record
dummy(0): luca-record2: before record: 3 or 4 to end
dummy0: digit 4...
dummy(0): luca-record2: after second record
```

The result is very like the previous one.

## 11.2 Collect

The *collect* instruction reads input from phone keyboard. The *collect* command has the form `collect count timeout term ignore` with:

- count: maximum number of digits to read, after count digits read, *collect* ends;
- timeout: maximum time pause between two following digits, after this timeout *collect* ends;
- term: digits to be used to end *collect* before count or timeout;
- ignore: digits to be ignored in the output of *collect*.

The output of the *collect* instruction is stored in the global variable *%session.digits*.

### Example 78: collect/1

*Testing environment: Ccscript 2.5.1, Bayonne 1.0.10, Bayonne 1.2.4*

**Source: luca-collect1.scr**

```
slog "before collect: * end, # ignored"
collect 10 15 "*" "#"
slog "after collect: digits " %session.digits
exit                  # NB: always an exit at the end
```

**Output: luca-collect1.out**

```
dummy(0): luca-collect1: before collect: * end, # ignored
dummy0: digit 1...
dummy0: digit 2...
dummy0: digit #...
dummy0: digit 3...
dummy0: digit 4...
dummy0: digit 5...
dummy0: digit #...
dummy0: digit 6...
dummy0: digit 7...
dummy0: digit *...
dummy(0): luca-collect1: after collect: digits 1234567
```

By the output it's to see that all pressed digits are stored in *%session.digits* but not # ignored and * used for term.

Every digits can be ignored or used to term:

### Example 79: collect/2

*Testing environment: Ccscript 2.5.1, Bayonne 1.0.10, Bayonne 1.2.4*

**Source: luca-collect2.scr**

```
slog "before collect: # end, 3,4 ignored"
collect 10 15 "#" "34"
slog "after collect: digits " %session.digits
exit                  # NB: always an exit at the end
```

**Output: luca-collect2.out**

```
dummy(0): luca-collect2: before collect: # end, 3,4 ignored
dummy0: digit 1...
dummy0: digit 2...
dummy0: digit 3...
dummy0: digit 4...
dummy0: digit 5...
dummy0: digit 6...
dummy0: digit *...
dummy0: digit #...
dummy(0): luca-collect2: after collect: digits 1256*
```

After "count" digits pressed the *collect* ends:

## Example 80: collect/3

*Testing environment: Ccscript 2.5.1, Bayonne 1.0.10, Bayonne 1.2.4*

**Source: luca-collect3.scr**

```
slog "before collect: max 10 digits"
collect 10 15 "*" "#"
slog "after collect: digits " %session.digits
exit                  # NB: always an exit at the end
```

**Output: luca-collect3.out**

```
dummy(0): luca-collect3: before collect: max 10 digits
dummy0: digit 1...
dummy0: digit 2...
dummy0: digit 3...
dummy0: digit 4...
dummy0: digit 5...
dummy0: digit 6...
dummy0: digit 7...
dummy0: digit 8...
dummy0: digit 9...
dummy0: digit 0...
dummy(0): luca-collect3: after collect: digits 1234567890
```

From the output you can see that the term digit is not pressed. Waiting "timeout" seconds between two digits the *collect* ends:

## Example 81: collect/4

*Testing environment: Ccscript 2.5.1, Bayonne 1.0.10, Bayonne 1.2.4*

**Source: luca-collect4.scr**

```
slog "before collect: max pause interdigit 5 secs"
collect 10 5 "*" "#"
slog "after collect: digits " %session.digits
exit               # NB: always an exit at the end
```

**Output: luca-collect4.out**

```
dummy(0): luca-collect4: before collect: max pause
          interdigit 5 secs
dummy0: digit 1...
dummy0: digit 2...
dummy0: digit 3...
dummy0: digit 4...
dummy(0): luca-collect4: after collect: digits 1234
```

From the output you can see that the term digit is not pressed and the maximum number of digits is not reached. Using two or more *collect*, digits in %session.digits are appended and digits of the first *collect* are counted in the second *collect* too.

## Example 82: collect, append

*Testing environment: Ccscript 2.5.1, Bayonne 1.0.10, Bayonne 1.2.4*

**Source: luca-collect5.scr**

```
slog "first  collect"
collect 5 10 "#" "*"
slog "after collect: digits " %session.digits
slog "second collect"
collect 8 10 "#" "*"
slog "after collect: digits (append) " %session.digits
exit               # NB: always an exit at the end
```

**Output: luca-collect5.out**

```
dummy(0): luca-collect5: first  collect
dummy0: digit 1...
dummy0: digit 2...
dummy0: digit 3...
dummy0: digit 4...
dummy0: digit #...
dummy(0): luca-collect5: after collect: digits 1234
dummy(0): luca-collect5: second collect
dummy0: digit 9...
dummy0: digit 8...
dummy0: digit 7...
dummy0: digit 6...
dummy(0): luca-collect5: after collect: digits (append) 12349876
```

The instruction *cleardigits* clears %session.digits (like `clear %session.digits`) and let using more *collect* without problems or digits appending:

## Example 83: collect, cleardigits

*Testing environment: Ccscript 2.5.1, Bayonne 1.0.10, Bayonne 1.2.4*

**Source: luca-collect6.scr**

```
slog "first  collect"
collect 5 10 "#" "*"
slog "after collect: digits " %session.digits
cleardigits
slog "digits cleared"
slog "second collect"
collect 8 10 "#" "*"
slog "after collect: digits " %session.digits
exit               # NB: always an exit at the end
```

**Output: luca-collect6.out**

```
dummy(0): luca-collect6: first  collect
dummy0: digit 1...
dummy0: digit 2...
dummy0: digit 3...
dummy0: digit 4...
dummy0: digit 5...
dummy(0): luca-collect6: after collect: digits 12345
dummy(0): luca-collect6: digits cleared
dummy(0): luca-collect6: second collect
dummy0: digit 8...
dummy0: digit 7...
dummy0: digit 6...
dummy0: digit 5...
dummy0: digit 4...
dummy0: digit 3...
dummy0: digit 2...
dummy0: digit 1...
dummy(0): luca-collect6: after collect: digits 87654321
```

## 11.3 Input with events

As seen in the event section, every phone digits can generate an event, so event handler can be used to read input. Digits values are still stored in %session.digits.

### Example 84: input event

*Testing environment: Ccscript 2.5.1, Bayonne 1.0.10, Bayonne 1.2.4*

**Source: luca-input-event.scr**

```
goto ::start
::start
  goto ::start  # infinite loop
^pound
  slog "digits " %session.digits
  exit               # NB: always an exit at the end
^dtmf
  goto ::start
```

**Output: luca-input-event.out**

```
dummy0: digit 1...
dummy0: digit 2...
dummy0: digit 3...
dummy0: digit 4...
dummy0: digit *...
dummy0: digit #...
dummy(0): luca-input-event::start: digits 1234*#
```

The output is very like *collect* behavior with # term digit (but in this case the # digit is stored in %session.digits).

# 12 TGI

The TGI module lets Bayonne to interact with external languages like Perl and Python.

The TGI call is made using *libexec* command in the form:
`libexec timeout outscript %param`
with

- timeout: maximum time waited by Bayonne for external script ending;
- outscript: external script invoked;
- %param: set of params (empty or with a lot of element) for the invoked script.

The default path for external script to be invoked is /usr/libexec/bayonne.

## 12.1 Perl

To exchange data with perl scripts, TGI has two instructions:

- $TGI::QUERY{'param'}: import the values of the *param* variables, this variable must be passed in the command line;
- TGI::set("ccscript_var",$perl_var): export the value to ccscript, sets a ccscript value from a perl value.

A basic perl script is the follow:

**luca-tgiperl1.pl**

```
#!/usr/bin/perl
# use lib '/usr/libexec/bayonne/';     # 1.0.x
use lib '/usr/local/libexec/bayonne/'; # 1.2.x
use TGI;
$fatt1 = $TGI::QUERY{'var1'};
$fatt2 = $TGI::QUERY{'var2'};
$result = $fatt1*$fatt2;  # any operation
TGI::set("res",$result);
exit
```

To use this perl script:

## Example 85: TGI perl

*Testing environment: Ccscript 2.5.1, Bayonne 1.0.10, Bayonne 1.2.4, perl 5.6.1*

**Source: luca-tgiperl1.scr**

```
set %var1 8
set %var2 7
slog.info %var1 " " %var2
libexec 10 luca-tgiperl1.pl %var1 %var2
slog.info %res
exit          # NB: always an exit at the end
```

**Output: luca-tgiperl1.out**

```
dummy(0): luca-tgiperl1: 8 7
tgi: cmd=luca-tgiperl1.pl query=var1=8&var2=7
     digits= clid=UNKNOWN dnid=UNKNOWN
fifo: cmd=wait 0 14160
fifo: cmd=SET&0&res&56
fifo: cmd=exit 0 0
dummy(0): luca-tgiperl1: 56
```

The variables %var1 and %var2 are passed to perl that return %res value.

In this second perl script variables are uppercase too:

**luca-tgiperl2.pl**

```
#!/usr/bin/perl
# use lib '/usr/libexec/bayonne/';     # 1.0.x
use lib '/usr/local/libexec/bayonne/'; # 1.2.x
use TGI;
$fatt1 = $TGI::QUERY{'Var1'};
$fatt2 = $TGI::QUERY{'Var2'};
$var3 = $TGI::QUERY{'var3'};
$result = $fatt1*$fatt2;  # any operation
TGI::set("Res",$result);
TGI::set("Res2",$var3);
TGI::set("result",$result);
TGI::set("result2",$var3);
exit
```

The Bayonne becomes:

## Example 86: TGI perl, uppercase/lowercase

*Testing environment: Ccscript 2.5.1, Bayonne 1.0.10, Bayonne 1.2.4, perl 5.6.1*

**Source: luca-tgiperl2.scr**

```
set %Var1 8
set %Var2 7
set %var3 10
slog.info %Var1 " " %Var2 " " %var3
libexec 20 luca-tgiperl2.pl %Var1 %Var2 %var3
```

```
slog.info "Res: "%Res
slog.info "Res2: "%Res2
slog.info "result: "%result
slog.info "result2: "%result2
exit          # NB: always an exit at the end
```

**Output: luca-tgiperl2.out**

```
dummy(0): luca-tgiperl2: 8 7 10
tgi: cmd=luca-tgiperl2.pl query=Var1=8&Var2=7&var3=10
     digits= clid=UNKNOWN dnid=UNKNOWN
fifo: cmd=wait 0 14183
fifo: cmd=SET&0&Res&0
fifo: cmd=SET&0&Res2&10
fifo: cmd=SET&0&result&0
fifo: cmd=SET&0&result2&10
fifo: cmd=exit 0 0
dummy(0): luca-tgiperl2: Res: 0
dummy(0): luca-tgiperl2: Res2: 10
dummy(0): luca-tgiperl2: result: 0
dummy(0): luca-tgiperl2: result2: 10
```

It's possible to see that:

- uppercase and lowercase variables are correctly exported from perl to ccscript;
- uppercase variables aren't imported correctly from ccscript to perl.

A perl script can need several seconds to compute its operation:

**luca-tgiperl3.pl**

```
#!/usr/bin/perl
# use lib '/usr/libexec/bayonne/';     # 1.0.x
use lib '/usr/local/libexec/bayonne/'; # 1.2.x
use TGI;
$fatt1 = $TGI::QUERY{'var1'};
$fatt2 = $TGI::QUERY{'var2'};
sleep(10);
$result = $fatt1*$fatt2;  # any operation
TGI::set("result",$result);
exit
```

The *libexec* command has a timeout, if timeout is greater than perl script execution time, the results are imported (and printed) correctly:

## Example 87: TGI perl, timeout/1

*Testing environment: Ccscript 2.5.1, Bayonne 1.0.10, Bayonne 1.2.4, perl 5.6.1*

**Source: luca-tgiperl3.scr**

```
set %var1 8
set %var2 7
slog.info %var1 " " %var2
libexec 20 luca-tgiperl3.pl %var1 %var2
```

```
slog.info %result
exit          # NB: always an exit at the end
```

**Output: luca-tgiperl3.out**

```
dummy(0): luca-tgiperl3: 8 7
tgi: cmd=luca-tgiperl3.pl query=var1=8&var2=7
     digits= clid=UNKNOWN dnid=UNKNOWN
fifo: cmd=wait 0 14193
fifo: cmd=SET&0&result&56
fifo: cmd=exit 0 0
dummy(0): luca-tgiperl3: 56
```

If timeout is lesser than perl script execution time, the results aren't imported (and printed) correctly:

### Example 88: TGI perl, timeout/2

*Testing environment: Ccscript 2.5.1, Bayonne 1.0.10, Bayonne 1.2.4, perl 5.6.1*

**Source: luca-tgiperl4.scr**

```
set %var1 8
set %var2 7
slog.info %var1 " " %var2
libexec 5 luca-tgiperl3.pl %var1 %var2
slog.info "result: "%result
exit          # NB: always an exit at the end
```

**Output: luca-tgiperl4.out**

```
dummy(0): luca-tgiperl4: 8 7
tgi: cmd=luca-tgiperl3.pl query=var1=8&var2=7
     digits= clid=UNKNOWN dnid=UNKNOWN
fifo: cmd=wait 0 14211
fifo: cmd=SET&0&result&56
dummy(0): luca-tgiperl4: result:
```

From output it's possible to see that the %result value is undefined because the perl script doesn't end before timeout.

## 13 Copyright