Chapitre 17

Éditeur de script en Lua

Dans la version 3.2 du logiciel pour ordinateur TI-Nspire CAS il est possible d'éditer des scripts en langage Lua et de les convertir en fichiers **.tns**. Ceci permet de programmer des applications et même des jeux… exécutables sur ordinateur comme sur unité nomade TI-Nspire CAS. La première partie de ce chapitre porte sur le langage Lua, la seconde sur l'éditeur de script du logiciel pour ordinateur TI-Nspire CAS ainsi que sur l'API TI-Nspire. Vous trouverez des informations sur Lua sur le site **http://www.lua.org/** et **http://lua-users.org/wiki/**, voir également pour la seconde partie de ce chapitre, les sites : **http://www.inspired-lua.org/** et **http://www.compasstech.com.au/**.

1. Le langage Lua

Lua est un langage interprété de très petite taille, il a été développé par Luiz Henrique de Figueiredo, Roberto Ierusalimschy et Waldemar Celes, membres du groupe de recherche TeCGraf, de l'université de Rio de Janeiro au Brésil en 1993 (Lua signifie Lune en portugais) pour être inclus comme langage de script dans des logiciels ou « embarqué ».

Lua est écrit en langage C ANSI strict, et de ce fait est compilable sur une grande variété de systèmes. Il tourne sur des systèmes aussi variés que Windows, Unix, PlayStation, XBox, Mac OS X, RISC OS, Symbian OS, PalmOS et dernièrement sur TI-Nspire.

C'est un langage simple avec peu de concepts (d'où sa petite taille) mais qui sont puissants et flexibles. Bien qu'interprété, il est très rapide.

Il est souvent utilisé dans des systèmes embarqués où sa compacité est très appréciée. Il profite de la compatibilité que possède le langage C avec un grand nombre de langages pour s'intégrer facilement dans la plupart des projets.

On trouve un usage important de Lua dans des environnements aussi variés que la conception de page Web, la robotique, la conception de jeux et même la composition de pages mathématiques (LuaTEX voir : **http://www.luatex.org/**).

1.1 *Premiers pas*

Les instructions en Lua peuvent se terminer par un point-virgule, mais ce n'est pas nécessaire, par contre si l'on met plusieurs instructions sur la même ligne, il est préférable pour des raisons de lisibilité de les séparer par des points-virgules.

Nous noterons les instructions isolées (le prompt (>) devant l'instruction n'est là que pour discerner les entrées et les sorties, il ne fait pas partie de la syntaxe) :

```
> print("Hello".." "..."World")
```

le résultat :

```
Hello World
```

Les ensembles élémentaires d'instructions (chunks) seront notés :

```
> for i = 2, 10, 2 do
>> print(i^2)
>> end
```

- **Commentaires**

Pour faire des commentaires sur une ligne on place devant le texte deux tirets (--) devant la ligne et sur plusieurs lignes, on utilise en plus des tirets, des doubles crochets. :

```
> --[[
>> print(" Lignes" )
>> print(" multiples." )
>> ]]
```

☞ *On peut aussi mettre deux tirets devant chaque ligne.*

- **Noms de variables**

Le nom d'une variable peut contenir des lettres, des chiffres, des _ (tiret bas), mais ne peut pas commencer par un chiffre. Le tiret bas peut être utilisé tout seul comme variable "muette". Lua différencie majuscules et minuscules : Ab, aB et AB seront considérées comme trois variables différentes.

Noms réservés en Lua : **`and`**, **`break`**, **`do`**, **`else`**, **`elseif`**, **`end`**, **`false`**, **`for`**, **`function`**, **`if`**, **`in`**, **`local`**, **`nil`**, **`not`**, **`or`**, **`repeat`**, **`return`**, **`then`**, **`true`**, **`until`**, **`while`**. Ils ne peuvent pas être utilisés comme nom de variables.

1.2 *Instructions*

Affectation

L'affectation d'une valeur à une variable se fait à l'aide du signe =. Une variable non affectée a pour valeur **`nil`**.

```
> a = 0.52
> b = "bonjour"
> compt = 1
> compt = compt + 1
```

On peut effacer le contenu d'une variable en tapant :

```
> b = nil
```

Affectations multiples :

Lua permet de faire des affectations multiples. Lors d'une affectation multiple, Lua commence par évaluer les valeurs à assigner et exécute ensuite les affectations.
Si le nombre de valeurs est inférieur au nombre de variables, les variables excédentaires reçoivent la valeur **`nil`**.

```
> c = 3
> a, b, c = 1, 2
```

```
> print(a,b,c)
1   2   nil
```

L'affectation multiple peut être utilisée pour permuter le contenu de deux variables.

```
> x = 1 ; y = 2
> x, y = y, x   -- échange les contenus de x et de y
> print(x,y)
2   1
```

Typage des variables

Il n'est pas besoin de déclarer le type d'une variable contrairement à certain langage. Le type de la variable est défini en fonction de la valeur ou de l'objet qui lui a été assigné ou des opérations effectuées. Ceci s'appelle du *typage dynamique*.

Lua étant un langage *réflexif*, il est possible d'utiliser la fonction **type**() pour obtenir le type d'une variable.

```
> a = "10"          -- a reçoit une chaîne de caractères
> print(a, type(a)) -- affiche la valeur de a et son type
10   string
> a = a + 1         -- ajoute un nombre à la chaîne et force son typage (coercition)
> print(a, type(a)) -- affiche de nouveau la valeur de a et son type
11   number
```

Les principaux types : *number*, *string*, *nil*, *boolean*, *function*, *table*, *userdata* and *thread*.

Le type *userdata* est fourni pour permettre à des données provenant du C d'être stockées dans des variables Lua. Nous n'en parlerons pas ici ainsi que du type *thread*.

Les principaux types :

```
> print(type(a))--a non affectée
nil
> a="a"
> print(type(a))
string
> a=1
> print(type(a))
number
> a=print
> print(type(a))
function
> a={}
> print(type(a))
table
```

Nombres (type *number*) représente les nombres réels, nombres à virgule flottante en double précision. Exemple : 2.35e-5 (2.35×10^{-5}).
Lua ne possède pas de type entier (*integer*).
La fonction **tonumber** permet de convertir une chaîne de caractères en nombre, le deuxième argument facultatif permet de choisir la base (entier entre 2 et 36 inclus).

```
> print(tonumber("010101"))
10101

> print(tonumber("010101",2))
21
```

Opérations : '+' (addition), '–' (soustraction), '*****' (multiplication), '**/**' (division), '^' (exponentiation), '%' (modulo), l'opérateur unaire '-' (négation).

```
> print((math.pi)^(-1/2)) ; print(27%5)

0.56418958354776
2
```

Chaînes de caractères (type *string*), les chaîne de caractères doivent être mises entre guillemets doubles ou simples ("bonjour" ou 'bonjour'). On peut insérer dans le texte des guillemets (\"), des apostrophes (\'), des retours lignes (\n), des tabulations (\t)…
La fonction **tostring** permet de convertir un argument de n'importe quel type en chaîne de caractères.

Bouléens (type *boolean*), le type booléen a uniquement deux valeurs possibles *true* et *false*, contrairement à des langages comme Maple (ou celui de TI-Nspire CAS), où il y a une troisième valeur *FAIL* (pour Maple).

Exemples :

```
> print(1 == 0)            -- (==) égal à

false

> print("a"<="b")          -- (<=) inférieur ou égal à

true

> print(nil ~= true)       -- (~=) différent de

true

> print(nil == false)

false
```

☞ *Les opérateurs de relation sont : ==, <, >, <=, >=, ~=.*

Opérateurs booléens : *and*, *or*, *not*.

Tables (type *table*), une table est une collection d'objets indexés par des clés qui peuvent être des nombres, des chaînes de caractères ou tout autre élément du langage Lua, excepté **`nil`**. Les différents types d'index peuvent être mélangés. Les tables n'ont pas besoin d'être déclarées, n'ont pas de dimension prédéfinie et peuvent être crées de façon dynamique.

C'est la structure de données principale de Lua, c'est en fait la seule.

Exemples :

```
> t = {clef1="val1", clef2="val2", clef3="val3"}[1]
> print(t["clef2"]) [2]
val2
```

L'opérateur # permet d'obtenir le nombre d'éléments de la table si elle est indexée par des entiers.

```
> Jours={"dimanche","lundi","mardi","mercredi","jeudi","vendredi","samedi"}
> print(#Jours)
7
> print(Jours[3])
mardi
```

Une liste donnant les cubes des 10 premiers entiers naturels :

```
> T3={}              -- on initialise à la table vide
> for k = 1, 10 do
>>  T3[k]=k^3
>> end
```

Autre exemple :

```
> T={A="a", "deux", C="c", "trois", D=4}
> print(T[1], T[2], T[3])
deux   trois   nil
> print(T.D, #T)
4  2
```

☞ *T[i] ne donne de résultat que si la clé est un entier, # ne donne que le nombre d'éléments de la table ayant pour clé un entier. T.D est un raccourci syntaxique de T["D"].*

On peut ajouter un élément à une table ou le supprimer :

```
> T[1] = nil ; T[3] = 13
> for i, t in pairs(T) do[3]
>>  print(i,t)
>> end
2    trois
A    a
D    4
C    c
3    13
```

☞ *L'ordre n'est pas forcément respecté.*

La fonction **unpack** permet d'obtenir les éléments d'une table indexée par des entiers.

```
print(unpack(T3))
```

[1] Il n'est pas nécessaire de mettre les clés entre guillemets lorsque se sont des chaînes de caractères.

[2] Par contre pour l'appel c'est nécessaire, sinon clef2 est évalué à **nil** et print(t[clef2]) retourne **nil**.

[3] La fonction **pairs** permet de parcourir les couples (*clé*, *valeur*) d'une table. La fonction **ipairs** permet de parcourir les tables indexées par des entiers (*array*).

```
1  8  27  64  125  216  343  512  729  1000
```

Création de matrices

Création de la matrice nulle à *n* lignes, *p* colonnes.

```
> mat = {}                -- crée la table (matrice)
> for i=1,n do
>>  mat[i] = {}            -- crée une nouvelle ligne
>>  for j=1,p do
>>    mat[i][j] = 0        -- remplit la ligne de 0
>>  end
>> end
```

Les tables en Lua étant des objets, vous devez créer chaque ligne de façon explicite pour créer une matrice.

L'opérateur # (longueur d'une chaîne de caractères)

```
> print(# " abcd" )
4
> print(# " \n" )[4]
1
```

Permet également, dans certains cas, de déterminer le nombre d'éléments d'une table comme nous l'avons vu précédemment. On peut aussi obtenir la longueur d'une chaîne de caractères à l'aide de la fonction **string.len** (marche également avec des chiffres, alors qu'avec # il faut impérativement les guillemets).

```
> print(string.len("Bonjour"))
7
> print(string.len(123))
3
> print(#123)
input:3: attempt to get length of a number value
```

Concatenation

L'opérateur de concaténation est noté dans Lua par (..) (deux points).

```
> print("Hello".." "..."World")
Hello World
```

Si l'un des opérandes est un nombre, Lua convertit ce nombre en chaîne de caractère (attention de mettre un espace entre le nombre et les deux points, sinon Lua interprète le nombre suivi du premier point comme un nombre décimal, ce qui provoque une erreur) :

```
> print(1 .. 0)
10
```

[4] **\n** qui permet d'insérer des changements de ligne lors du formatage d'un texte n'est considéré que pour un caractère.

```
> print(type(1 .. 0))
string
```

1.3 *Fonctions*

Définir des fonctions

Les fonctions peuvent être définies en utilisant le mot-clé **`function`**. Il y a deux façons de déclarer des fonctions :

`function` *nom_fonction (argument) corps* **end**

ou

nom_fonction **= `function(`** *arguments*) corps **end**

Exemple d'une fonction simple permettant d'élever un nombre au carré :

```
> function carre(x) return x^2 end
> print(carre(7))
49
```

Nombre variable d'arguments

Il est possible d'écrire des fonctions pouvant être appelées avec un nombre variable d'arguments. Les trois points (...) dans la liste des paramètres indiquent que la fonction accepte un nombre variable d'arguments.

Exemple, fonction calculant la somme d'un nombre variable d'entiers :

```
> function add (...)
>> local s = 0
>> for i, x in ipairs{...} do
>>  s = s + x
>> end
>> return s
>> end
```

Utilisation :

```
> print(add(3, 4, 10, 25, 12))
54
```

Variables locales

Lua permet de déclarer des variables locales, il suffit de les créer avec l'instruction **local**.

```
> i = 1              -- variable globale
> local j = 2        -- variable locale
```

Déclaration au sein d'une fonction :

```
> a=1                -- variable globale
```

```
> function essai()
>> local a=2        -- variable locale
>> b=2              -- variable globale
>> print(a,b)
>> end

> essai()

2   2

> print(a,b)

1   2
```

☞ *La fonction* ***print*** *retourne ici les valeurs des variables globales a et b, alors que la fonction* ***essai*** *retourne la valeur du a local à la fonction et la valeur du b global.*

1.4 *Structure conditionnelle*

La structure conditionnelle de base, lorsqu'il n'y a que deux éventualités, est :

if *condition* **then** *instructions*
else *autres instructions*
end

avec sa forme simplifiée, si l'on ne fait rien lorsque la condition n'est pas remplie:

if *condition* **then** *instructions* **end**

La syntaxe de la structure conditionnelle complète est donnée par :

if *condition* **then** *instructions*
elseif *autre condition* **then** *autres instructions*
⋮
elseif *autre condition* **then** *autres instructions*
else *autres instructions*
end

Le **end** final permet d'indiquer la fin du bloc constituant la structure conditionnelle. Les **elseif** facilitent la programmation dans le cas d'alternatives multiples.

Exemple. Définition de la fonction définie par :

$$f(i,j)=\begin{cases} j & \text{si } i>j \\ i & \text{si } j>i \\ 1 & \text{si } i=j \end{cases}$$

```
>  function f(i,j)
>> if       i>j then return j
>> elseif   i<j then return i
>> else     return 1
```

```
>> end
>> end
```

1.5 *L'itération*

- **Boucle for avec index numérique**

La syntaxe est donnée par :

> **for** *variable* = *début* **,** *fin* **,** *pas* **do**
>
> *instructions*
>
> **end**

☞ pas *est facultatif, valeur par défaut 1 ;* pas *peut être négatif, mais alors,* début *doit être supérieur à* fin.

Exemple :

Affichage des carrés des entiers naturels pairs inférieurs à 10.

```
> for i = 2, 10, 2 do
>> print(i^2)
>> end
```

- **Boucle for générique**

> **for** *var-list* **in** *expr-list* **do**
>
> *instructions*
>
> **end**

où *var-list* est une liste d'une ou plusieurs variables séparées par des virgules et *expr-list* une liste d'une ou plusieurs expressions séparées par des virgules.

Exemples :

Affichage des éléments d'un tableau.

```
> for i,x in ipairs(t) do print(x) end
```
[5]

Affichage du maximum des termes d'une liste et de la place du maximum dans la liste :

```
> function maximum(x)
>> local im = 1             -- index de la valeur maximale
>> local m = x[im]          -- valeur maximale
>> for i,val in ipairs(x) do
>>   if val > m then
>>   im = i; m = val
>>   end                     -- fin du if
>> end                      -- fin du for
>> return m, im             -- on renvoit 2 valeurs
>> end

> print(maximum({12,9,18,11,25,13,7,25}))
```

[5] La fonction **ipairs** permet de parcourir les tables indexées par des entiers (*array*).

```
25  5
```

☞ *Il faut passer le résultat de la fonction comme argument à* ***print*** *pour afficher le résultat.*
L'indice affiché est le premier indice trouvé dans le cas de plusieurs valeurs maximales.

- **Boucle While**

Les instructions sont effectuées tant que la condition reste vraie… on prendra garde de bien vérifier que les instructions modifient la condition et qu'elle devienne fausse, sinon on rentre dans une boucle infinie.

La syntaxe de l'itération est donnée par :

while *condition* **do**
  instructions
end

Exemple :

Affichage des carrés des entiers naturels pairs inférieurs à 10 (seconde version).

```
> i = 2

> while i <= 10 do
>> print(i^2)
>> i = i + 2
>> end
```

- **Repeat**

Les instructions sont répétées tant que la condition est fausse (prendra garde, ici aussi, à ne pas entrer dans une boucle infinie).

La syntaxe de l'itération est donnée par :

repeat
  instructions
until *condition*

☞ *Ici la boucle* est *effectuée au moins une fois.*

Exemple :

Calcul des factorielles des entiers naturels jusqu'à l'obtention d'une factorielle supérieure à 100.

```
> n, fac = 1, 1                    -- Initialisation des variables
> repeat
>> print(n .. " ! = " .. fac)
>> n = n + 1                        -- Incrémente n
>> fac = fac * n                    -- Calcule sa factorielle
>> until fac >= 100

1! = 1
2! = 2
3! = 6
4! = 24
```

- **Break**

L'instruction **`break`** rencontrée dans une boucle **`while`**, **`for`** ou **`repeat`** provoque la sortie immédiate de la boucle la plus interne contenant le **`break`**. On peut l'utiliser par exemple pour interrompre une recherche dès que l'on a trouvé le terme désiré.

Exemple :

```
> t={"rouge","vert","blanc","jaune"}
> print(#t)
4
> for i,x in ipairs(t) do
>> print(i,x)
>> if x=="blanc" then break
>> end
>> end
1   rouge
2   vert
3   blanc
```

Valeurs retournées par une fonction

Lua ajuste toujours le nombre de résultats d'une fonction aux circonstances de l'appel. Quand nous appelons une fonction comme une instruction, Lua ne retourne aucun résultat. Quand nous utilisons un appel comme une expression, Lua garde seulement le premier résultat, sauf quand l'appel est la dernière (ou la seule) expression dans une liste d'expressions, auquel cas tous les résultats sont obtenus.

Ces listes apparaissent dans quatre constructions dans Lua :

- affectations multiples,
- arguments d'une fonction,
- constructeurs de table,
- utilisation de **`return`**.

Pour illustrer ces cas, nous utiliserons la fonction suivante :

```
> function f () return "a","b" end -- retourne 2 résultats
> f ()        -- ne donne rien, il faut composer avec print, voir plus bas
```

Dans une affectation multiple, un appel de fonction en tant que dernière (ou unique) expression, produit autant de résultats que nécessaire pour correspondre au nombre de variables :

```
> x,y = f()          -- x="a", y="b"
> x,y,z = 1, f()     -- x=1, y="a", z="b"
```

Si une fonction ne retourne pas de résultat ou en retourne un nombre insuffisant, Lua remplace les valeurs manquantes par des **`nil`** :

```
x,y,z = f()          -- x="a", y="b", z=nil
```

Un appel de fonction qui n'est pas le dernier élément dans la liste, produit toujours exactement un résultat :

```
x,y,z = f(), 1        -- x="a", y=1, z=nil
```

Lorsqu'un appel de fonction est le dernier (ou l'unique) argument d'un autre appel, tous les résultats du premier appel sont pris comme arguments, sinon seul le premier est pris. Nous avons déjà vu des exemples de cette construction avec la fonction **print** :

```
> print(f())
a   b

> print(f(), 1)
a   1

> print(f() .. "x")
ax
```

On a le même comportement avec les constructeurs de tables :

```
> t1={f(),"c"}
> print(t1[1],t1[2],t1[3])

a   c   nil

> t2={"c", f()}
> print(t2[1],t2[2],t2[3])

c   a   b
```

Vous pouvez forcer un appel à une fonction pour qu'elle retourne exactement un seul résultat en l'incluant dans une paire supplémentaire de parenthèses :

```
> print((f()))

a
```

1.6 *Programmation récursive*

Lua permet l'écriture de fonction récursive, en voici un exemple classique : la fonction factorielle.

```
> function factorielle(n)
>> assert(n==math.floor(n) and n>=0, n .. " n'est pas un entier naturel")
>> if n==0 then return 1
>> else return n*factorielle(n-1)
>> end
>> end
```

☞ *La fonction* **assert** *retourne son second argument (le message entre guillemets) si la condition passée en premier argument est fausse.*

Un deuxième exemple : fonction permettant de mettre à plat des listes imbriquées.

```
> function flatten(liste)
>> if type(liste) ~= "table" then return {liste} end
>> local L = {}
>> for _, elem in ipairs(liste) do
>>  for _, val in ipairs(flatten(elem)) do
>>   L[#L + 1] = val
>>  end
>> end
>> return L
```

```
>> end
```

L'instruction **L[#L + 1] = val** équivaut à **table.insert(L, val)**.

table.insert(list [, n], val) ajoute *val* à la liste *list* en *n*-ième position et décale les éléments suivants. En l'absence du deuxième argument *n* (entier naturel) l'ajout est fait en fin de liste.

Le tiret bas dans « les **for** » représente une variable muette, on n'a pas besoin de la nommer car elle n'intervient pas dans les calculs suivants.

Utilisation:

```
> liste_test = {{1}, 2, {{3,4}, 5}, {{{}}}, {{{6}}}, 7, 8, {9,10}, {}}

> print(table.concat(flatten(liste_test), ", "))

1, 2, 3, 4, 5, 6, 7, 8, 9, 10
```

☞ *La fonction **table.concat** permet d'obtenir les éléments d'une liste séparés par un séparateur (second argument facultatif), s'il est omis le séparateur est un espace.*

1.7 *Les bibliothèques*

Nous avons déjà rencontré dans les pages précédentes les fonctions math.pi, math.floor ou table.insert, ces fonctions proviennent de bibliothèques, les deux premières de la bibliothèque mathématiques.

Bibliothèque mathématique

Elle comprend les fonctions suivantes :

math.abs	math.acos	math.asin	math.atan	math.atan2	math.ceil
math.cos	math.cosh	math.deg	math.exp	math.floor	math.fmod
math.frexp	math.huge	math.ldexp	math.log	math.log10	math.max
math.min	math.modf	math.pi	math.pow	math.rad	math.random
math.sin	math.sinh	math.sqrt	math.tanh	math.tan	math.randomseed

Les fonctions trigonométriques travaillent en radians, pour passer de radian en degré on peut utiliser math.deg et réciproquement math.rad pour passer de degré en radian.

math.random génère un nombre pseudo aléatoire. Elle peut être appelée de trois façons différentes :

- sans argument : donne un nombre entre 0 et 1,
- un argument entier *n* : donne un entier entre 1 et *n*
- deux arguments entiers *n* et *m* : donne un entier entre *n* et *m*

```
> print(math.random()) ; print(math.random(6)) ; print(math.random(10,100))

0.84018771715471
3
81
```

Le générateur peut être réinitialisé à l'aide de math.randomseed, une bonne façon est d'utiliser :

```
> math.randomseed(os.time))
```

os.time fait partie de la **bibliothèque I/O** que nous n'aborderons pas ici, elle retourne un nombre de secondes correspondant à un temps écoulé à partir d'une date donnée.

Bibliothèque table

Nous avons déjà rencontré deux fonctions de cette bibliothèque : table.insert et table.concat.

Sitons également table.remove(tab [, pos]) qui permet de récupérer l'élément d'une table en position *pos*, si le deuxième argument est omis, c'est le dernier élément qui est retourné. Enfin table.sort permet d'ordonner une table.

Bibliothèque chaîne de caractères

Nous avons déjà vu une fonction de cette bibliothèque en page 6 : string.len.

Les fonctions string.char et string.byte convertissent les chaînes de caractères en leur code numérique ASCII et réciproquement.

string.format permet de formater du texte.

string.sub(text, i [, j]) retourne une sous chaîne de text allant du i-ième terme au j-ième (dernier terme si j est omis), la chaîne text n'est pas modifiée ($j=-2$, donne l'avant dernier terme).

Il existe d'autres fonctions dans cette bibliothèque que nous ne citerons pas, voir l'aide de Lua.

Les deux dernières bibliothèques portent sur le système d'exploitation et le débogage.

1.8 *Métatables et métaméthodes*

Les tables sont une structure Lua qui permet de stocker des variables indexées par une clef (comme en PHP par exemple). Les tables servent à beaucoup de choses en Lua, c'est par exemple la structure utilisée pour représenter des pseudo-classes et objets.

Les métatables sont des tables pour lesquelles on a associé une fonction à chaque opérateur de base (+, -, (), [], ...). Ce mécanisme est très proche de la redéfinition des opérateurs en C++. Par exemple, si la variable x contient une table associée à une métatable appropriée, l'appel de fonction x(arguments) sera valide car Lua cherchera alors dans la métatable comment traiter cet appel.

C'est ainsi qu'on peut implémenter l'héritage entre tables. Si un champ n'est pas trouvé lorsqu'il est demandé, Lua cherchera dans la métatable quelle table est parente et demandera à la table parente de fournir l'élément approprié.

En général, à chaque objet dans Lua est attaché un jeu d'opérations applicables à ce type d'objet. Nous pouvons ajouter des nombres, nous pouvons concaténer des chaînes de caractères... mais nous ne pouvons pas ajouter ou concaténer des tables, comparer des fonctions...

```
> print(1 .. 2) ; print("a".."b")
12
ab
```

Lua sait concaténer des nombres ou des chaînes de caractères.

Par contre, on ne peut pas concaténer deux tables.

```
> t1,t2={},{}
> print(t1 .. t2)
input:2: attempt to concatenate global 't1' (a table value)
```

Les métatables nous permettent de changer le comportement d'une valeur lorsqu'elle est soumise à une opération donnée. Par exemple, à l'aide de métatables, nous pouvons apprendre à Lua comment concaténer deux tables.

Chaque fois que l'on demande à Lua de concaténer deux tables, il vérifie si l'une d'elle, possède une métatable et si cette métatable a un champ __concat (deux tirets bas devant le nom). Si Lua trouve un champ __concat, il appelle ce champ qui est en fait une fonction appelée **métaméthode** pour effectuer l'opération. Il suffit que l'une des tables possède une telle métatable avec sa métaméthode pour que l'opération puisse se faire.

Chaque valeur dans Lua peut avoir une métatable. Les tables ont des métatables individuelles ; les valeurs d'autres types partagent une seule métatable commune à ce type, voir l'exemple ci-dessous pour des chaînes de caractères.

```
> print(getmetatable("Bonjour"))
table: 0x16058c0

> print(getmetatable("Hi"))
table: 0x16058c0
```

Lors de la création d'une table, Lua ne crée pas de métatable attachée. La fonction **getmetatable** retourne **nil** si l'objet passé en argument n'a pas de métatable, sinon elle retourne la métatable.

```
> t = {1,2,3}
> print(getmetatable(t))

nil
```

La fonction **setmetatable** permet de remplacer ou de créer, si elle n'existe pas, la métatable de la table passée en argument. On ne peut le faire que pour les tables, pour les autres types d'objets, le changement de métatable nécessite le passage par le C.

```
> mt = {}
> setmetatable(t, mt)
> print(getmetatable(t))

table: 0x15b8b20
```

Un exemple

Le langage Lua n'a pas de type *set* (ensemble), le seul type de données, comme on l'a vu, est le type *table*. Nous allons définir à partir de tables des objets dont on pourra définir l'union, l'intersection…

```
> Set = {}
> local mt={}
-- crée un ensemble avec les éléments de la liste passée en argument
> function Set.cons (l)
>> local set = {}
>> setmetatable(set, mt) -- définit la même métatable pour tous les ensembles
>> for _, v in ipairs(l) do set[v] = true end
>> return set
>> end
```

On définit l'union :

```
> function Set.union (a, b)
>> local r = Set.cons{}
>> for k in pairs(a) do r[k] = true end
>> for k in pairs(b) do r[k] = true end
>> return r
>> end
```

On définit l'intersection :

```
> function Set.intersection (a, b)
>> local r = Set.cons{}
>> for k in pairs(a) do
>>   r[k] = b[k]
>> end
>> return r
>> end
```

Pour pouvoir contrôler les calculs, nous allons écrire une fonction permettant d'afficher les ensembles.

```
> function Set.tostring (set)
>> local l = {}                -- liste pour recevoir tous les éléments de l' ensemble
>> for el in pairs(set) do
>>   table.insert(l, el)
>> end
>> return "{" .. table.concat(l, ", ") .. "}"
>> end
> function Set.print (s)
>> print(Set.tostring(s))
>> end
```

Ensuite on rajoute à la métatable les métaméthodes :

```
> mt.__add = Set.union
> mt.__mul = Set.intersection
> mt.__tostring = Set.tostring
```

Pour tester notre nouvel « objet », il ne reste plus qu'à définir deux ensembles et à en calculer l'union et l'intersection.

```
> ens1 = Set.cons{"a", "b", "c", "a", "d"}
> ens2 = Set.cons{"b", "d", "e", "f", "e", "g"}
> ens3 = ens1 + ens2
> ens4 = ens1 * ens2
> Set.print(ens3)
```

{a, c, b, e, d, g, f}

```
> Set.print(ens4)
```

{b, d}

On peut également protéger le nouvel objet (set) en protégeant sa métatable, afin qu'un utilisateur ne puissent ni l'éditer, ni la modifier. Si vous mettez un champ __metatable dans la métatable, **getmetatable** retourne la valeur du champ, tandis que **setmetatable** retourne une erreur :

```
> mt.__metatable = "Métatable protégée"
```

On obtient alors :

```
> print(getmetatable(ens1))
```

Métatable protégée

```
> setmetatable(ens1, {})
input:48: cannot change a protected metatable
```

Il existe des noms de fonctions prédéfinies, à placer dans les métatables, et qui correspondent aux opérateurs arithmétiques.

Tous ces noms s'écrivent avec deux tirets bas (__) suivi du nom.

- l'opérateur + correspond à la métaméthode __add()
- l'opérateur - correspond à la métaméthode __sub() (opérateur binaire)
- l'opérateur - correspond à la métaméthode __unm() (opérateur unaire)
- l'opérateur * correspond à la métaméthode __mul()
- l'opérateur / correspond à la métaméthode __div()
- l'opérateur % correspond à la métaméthode __mod()
- l'opérateur ^ correspond à la métaméthode __pow()
- l'opérateur # correspond à la métaméthode __len()
- l'opérateur < correspond à la métaméthode __lt()
- l'opérateur <= correspond à la métaméthode __le()
- l'opérateur == correspond à la métaméthode __eq()

☞ *Il n'y a pas de métaméthodes pour les opérateurs "~=", ">".et ">="*
Lua traduit "a ~= b" par "not a == b" ; a > b" par "not a <= b" ; "a >= b" par "not a < b"

Nous avons dans l'exemple précédent des métaméthodes permettant d'effectuer des opérations arithmétiques, il est également possible grâce aux métaméthodes de donner un sens à l'égalité, la différence, aux relations d'ordre… entre ensembles, voir pour cela et pour approfondir les notions de métatables et de métaméthodes : **Programming in Lua de Roberto Ierusalimschy**.

La métaméthode __index

Une métaméthode intéressante est __index. Supposons que l'on veuille obtenir le nombre d'éléments d'un ensemble.

```
>.print(card(ens1))
input:49: attempt to call global 'card' (a nil value)
```

Lua commence à chercher l'existence d'un index et si dans cet index figure un champ card. Si c'est le cas, il appelle la méthode et retourne le résultat, sinon on obtient un message d'erreur comme ci-dessus.

```
> mt.__index = function (s,k)
>> if k == "card" then return (string.len(Set.tostring(s)))/3 end
>> end
```

On peut maintenant obtenir le nombre d'éléments d'un ensemble :

```
print(ens1.card)
4
```

Valeurs par défaut dans une table

La valeur par défault de n'importe quel champ dans une table est **nil**. Il est facile de changer cela à l'aide d'une métatable et de la métaméthode __index.

```
> function valDefault (t, d)
>> local mt = {__index = function () return d end}
>> setmetatable(t, mt)
>> end
> tab = {x=10, y=20}
> print(tab.x, tab.z)
10  nil
> valDefault(tab, 0)
> print(tab.x, tab.z)
10  0
```

La métaméthode __newindex

Comme **__index** permet de gérer l'accès au contenu d'une table, **__newindex** permet de gérer la modification du contenu d'une table.

Prenons un exemple simple :

```
> local t = {}
> t.x = 1
> print(t.x)
1
```

Nous créons une table vide, nous créons un champ "x" auquel nous affectons la valeur "1", qui est ensuite affichée. À l'exécution, voici comment Lua interprète ce code. Après avoir initialisé la table **t**, vide, il tente d'y rajouter un nouveau champ. Pour cela, il cherche d'abord si la table possède une métatable. Si c'est le cas, il cherche la métaméthode **__newindex** et l'exécute en lui passant trois arguments :

- la table appelante
- le champ à créer
- la valeur à assigner

Si la table appelante ne possède pas de métatable, Lua se contente de créer ce champ dans la table et de faire l'assignation.

Nous allons concevoir des tables en lecture seule. L'idée, c'est de pouvoir empêcher toute modification d'un élément présent dans la table, et d'empêcher tout rajout d'une nouvelle entrée.

Les métaméthodes **__index** et **__newindex** sont automatiquement utilisées par Lua lorsque le champ recherché dans une table est manquant. Par contre si nous désirons empêcher la modification d'un champ existant dans une table, **__index** et **__newindex** ne nous servent à rien, puisque le champ que nous voulons modifier existe déjà dans la table. L'interpréteur n'aura donc pas à consulter les métaméthodes, puisque ce qu'il cherche est dans la table appelante.

On va donc travailler sur une table vide, le contenu de la table à protéger étant stocké dans une table miroir.

Si l'on définit une métatable pour la table à protéger, on peut parfaitement se servir des métaméthodes **__index** et **__newindex**.

```
> function Protect(t)
>> local tab = {}
```

```
>> local mt = { __index = t,
>>              __newindex = function (t, k, v)
>>              error("Table en lecture seule !")
>>              end }
>> setmetatable(tab, mt)
>> return tab
>> end
-- on crée une liste de constantes physiques
> local cst_phys = { g = 9.81, N = 6.02E23, e = 1.6E-19, k = 0.082}
> cst_phys = Protect(cst_phys) -- on protège la table
> print(cst_phys.g) -- on peut accéder à un champ de la table protégée
9.81
> cst_phys.g = 10  -- on ne peut ni modifier, ni ajouter de champ
input:18: Table en lecture seule !
```

Même résultat avec la tentative de création d'une nouvelle entrée :

```
> cst_phys.h=6.626E-34
```

1.9 *Programmation orientée objet*[6]

Lua n'a pas la notion "de classes" incorporée pour l'utilisation dans la programmation orientée objet comme certain langage de programmation comme par exemple C++ ou Java. Cependant, nous pouvons simuler beaucoup de caractéristiques des classes d'autres langues en utilisant juste les tables et les métaméthodes dont nous avons parlé dans le paragraphe précédent. Il est également possible de simuler la notion d'héritage et même d'héritage multiple (une classe peut hériter des propriétés de plusieurs classes mères, nous n'en parlerons pas ici). On n'a pas par contre les notions de **public**, et de **privé** comme en C++ par exemple.

Une table dans Lua est un objet avec une identité (un *self*), des valeurs, des opérations :

```
> Compte = {solde = 0}
> function Compte.retrait (r)
>> Compte.solde = Compte.solde - r
>> end
```

Cette définition crée une nouvelle fonction et la stocke dans le champ retrait de l'objet Compte. On peut alors l'appeler ainsi :

```
> Compte.retrait(100)
```

Cette façon de programmer la fonction ne permet pas de définir une *méthode* applicable à d'autres objets. La fonction ne marchera qu'avec cet objet particulier et ne sera plus utilisable si l'objet change de nom ou disparaît.

Pour créer une méthode, il faut agir sur le récepteur de l'opération. Pour cela, notre méthode a besoin d'un paramètre supplémentaire prenant la valeur du récepteur.
Ce paramètre est nommé d'habitude *self* :

```
> function Compte.retrait (self, r)
>> self.solde = self.solde - r
```

[6] Les exemples de ce paragraphe sont tirés du livre de Roberto Ierusalimschy : Programming in Lua.

```
>> end
```

la méthode est alors appelée par :

```
> a:retrait(100)
```

L'effet des deux-points est d'ajouter le paramètre *self* dans une définition de méthode ou dans un appel de méthode. Les deux-points sont seulement une facilité syntaxique. Nous pouvons définir une fonction avec la syntaxe (point) et l'appeler avec la syntaxe deux-points, ou vice versa, tant que nous manipulons le paramètre supplémentaire *self* correctement :

```
> Compte = { solde=0,
>> retrait = function (self, r)       -- self explicite
>> self.solde = self.solde - r
>> end
>> }
> function Compte:depot (d)       -- raccourcis syntaxique, self implicite
>> self.solde = self.solde + d
>> end
```

Les appels :

```
> Compte.depot(Compte, 200)
> Compte:retrait(100)             -- équivaut à Compte.retrait(Compte, 100)
```

Nos objets ont une identité, un état et des opérations sur cet état. Il manque un système de classes et la notion d'héritage.

Classes

Une classe fonctionne comme un « modèle » pour la création d'objets qui vont tous avoir le même comportement. Plusieurs langages orientés objet offrent le concept de classe, dans de tels langages, chaque objet est une instance d'une classe spécifique. Lua n'a pas le concept de classe, chaque objet définit son propre comportement et son identité propre.

Pour représenter une classe nous créons simplement un objet qui sera utilisé exclusivement comme un « modèle » pour d'autres objets (ses instances). Les classes permettent de stocker le comportement partagé par plusieurs objets.

Dans Lua, il est très simple de mettre en œuvre des « modèle », en utilisant l'idée d'héritage qui découle de l'utilisation des métatables vue dans le paragraphe précédent. Plus spécifiquement, si nous avons deux objets a et b, pour faire de b un « modèle » pour a, il suffit de poser :

```
> setmetatable(a, {__index = b})
```

Après cela, a va chercher dans b les opérations qu'il n'a pas. b peut être considéré comme une classe d'objets.

Revenons à notre exemple de compte bancaire. Pour créer d'autres comptes avec un comportement semblable à Compte, on fait en sorte que ces nouveaux objets héritent leurs opérations du « modèle » Compte, en utilisant la métaméthode __index. Il n'est pas nécessaire de créer une table supplémentaire pour être la métatable des objets de la classe Compte, il suffit d'utiliser la table de Compte elle-même :

```
> function Compte:new (t)
>> t = t or {}        -- crée une table vide si l' utilisateur n' en fournit pas une
>> setmetatable(t, self)
```

```
>> self.__index = self
>> return t
>> end
```

(Quand nous appelons Compte:new, *self* est égal à Compte ; ainsi, nous aurions pu utiliser Compte directement, au lieu de *self*. Cependant, l'utilisation de *self* servira pour introduire la notion d'héritage de classe dans la section suivante. Après ce code, qu'arrive-t-il quand nous créons un nouveau compte et que l'on appelle une méthode sur lui ?

```
> a = Compte:new{solde = 0}
> a:depot(100)
```

Quand nous créons le nouveau compte, a possède pour métatable Compte (*self* dans l'appel à Compte:new). Alors, quand nous appelons a:depot(100), nous appelons en réalité a.depot(a, 100). Cependant, Lua ne peut pas trouver une entrée depot dans la table a, ainsi, il examine l'entrée __index de la métatable.

La métatable de "a" est Compte et Compte. __index est aussi Compte (parce que la nouvelle méthode a fait self. __index=self). Donc, l'expression précédente se réduit à : Compte.depot(a, 100).

C'est-à-dire Lua appelle la fonction depot originale, mais en passant a comme paramétre *self*. Ainsi, le nouveau compte a hérité de la fonction depot de Compte. Par le même mécanisme, il peut hériter de tous les champs de Compte.

L'héritage marche non seulement pour des méthodes, mais aussi pour d'autres champs qui sont absents dans le nouveau compte. Une classe peut donc fournir non seulement des méthodes, mais peut donner aussi des valeurs par défaut pour les champs de l'instance. Dans la première définition de Compte, nous avons fourni un solde de valeur 0. Ainsi, si nous créons un nouveau compte sans un solde initial, il héritera de cette valeur par défaut :

```
> b = Compte:new()
> print(b.solde)
0
```

Quand nous appelons la méthode `depot` sur b, cela marche de la façon suivante :

```
b.solde = b.solde + v
```

(Parce que *self* vaut b). L'expression b.solde est évaluée à zéro et un dépôt initial est assigné à b.solde. Des accès ultérieurs à b.solde n'invoqueront pas la métaméthode index parce que maintenant b a son propre champ solde.

Héritage

Les classes étant des objets, elles peuvent hériter des méthodes d'autres classes. Ce comportement rend la notion d'héritage, au sens habituel de la programmation orientée objet, tout à fait facile à mettre en œuvre dans Lua.

Supposons que nous ayons une classe de base Compte :

```
Compte = {solde = 0}
function Compte:new (t)
t = t or {}
setmetatable(t, self)
self.__index = self
```

```
return t
end
function Compte:depot (d)
self.solde = self.solde + d
end
function Compte:retrait (r)
if r > self.solde then error "fonds insuffisants !" end
self.solde = self.solde − r
end
```

À partir de cette classe, nous voulons définir une classe fille CompteSpecial qui permet au client de retirer une somme supérieure au solde. Nous commençons par définir une classe vide qui hérite simplement de toutes les opérations de sa classe mère :

```
CompteSpecial = Compte:new()
```

Jusqu'ici, CompteSpecial n'est juste qu'une instance de Compte. Là où les choses changent c'est lorsque l'on définit une nouvelle table comme ceci par exemple :

```
s = CompteSpecial:new{limit=1000}
```

CompteSpecial hérite du constructeur new de Compte comme de toutes ses autres méthodes. Cette fois, cependant, quand new s'exécute, son paramètre *self* se réfère à CompteSpecial et non plus à Compte.

Donc, la métatable de s sera CompteSpecial, dont la valeur de l'entrée __index sera également CompteSpecial. Ainsi, s hérite de CompteSpecial, qui hérite de Compte.

Quand nous évaluons :

```
s:depot(100)
```

Lua ne peut pas trouver depot dans s, donc il regarde dans CompteSpecial, il n'y trouve pas non plus un champ depot, il regarde donc dans Compte et là il trouve la définition originale de depot.

Ce qui est spécial dans CompteSpecial, c'est que nous pouvons redéfinir n'importe quelle méthode héritée de sa classe mère, il suffit pour cela d'écrire la nouvelle méthode :

```
> function CompteSpecial:retrait (r)
>> if r − self.solde >= self:getLimit() then
>> error"Vous dépassez les bornes !… "
>> end
>> self.solde = self.solde − r
>> end
>> function CompteSpecial:getLimit ()
>> return self.limit or 0
>> end
```

Maintenant, quand nous appelons s:retrait (200), Lua ne va pas chercher dans Compte, parce qu'il trouve la nouvelle méthode retrait en premier dans CompteSpecial. La valeur de s.limit étant de 1000 (fixé lors de la création), le programme fait le retrait, laissant s avec un solde négatif.

Un aspect intéressant des objets dans Lua est que vous ne devez pas créer une nouvelle classe pour spécifier un nouveau comportement. Si seulement un objet a besoin d'un comportement spécifique, vous pouvez mettre en œuvre ce comportement directement dans l'objet. Par exemple, si le compte s représente un certain client spécial dont la limite est toujours 10 % de son solde, vous pouvez modifier seulement ce compte :

```
> function s:getLimit ()
>> return self.solde * 0.10
>> end
```

Après cette déclaration, l'appel s:retrait (200) lance la méthode retrait de CompteSpecial, mais quand retrait fait appel à self:getLimit, c'est cette dernière définition qu'il invoque.

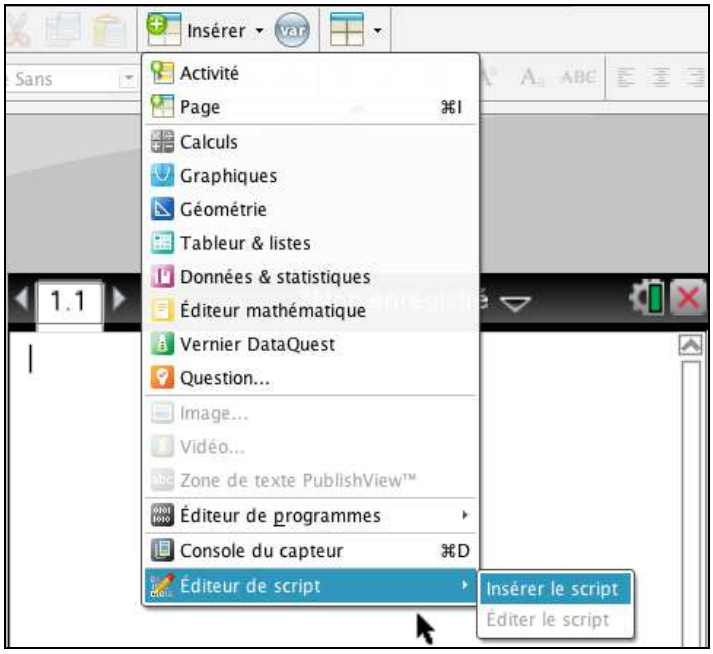
Dans ce survol du langage Lua, j'ai insisté sur l'utilisation des tables car c'est une notion fondamentale. Pour approfondir d'autres aspects du langage je vous renvoie au livre de Roberto Ierusalimschy.

Nous allons maintenant voir la mise en œuvre de ce langage dans le logiciel TI-Nspire.

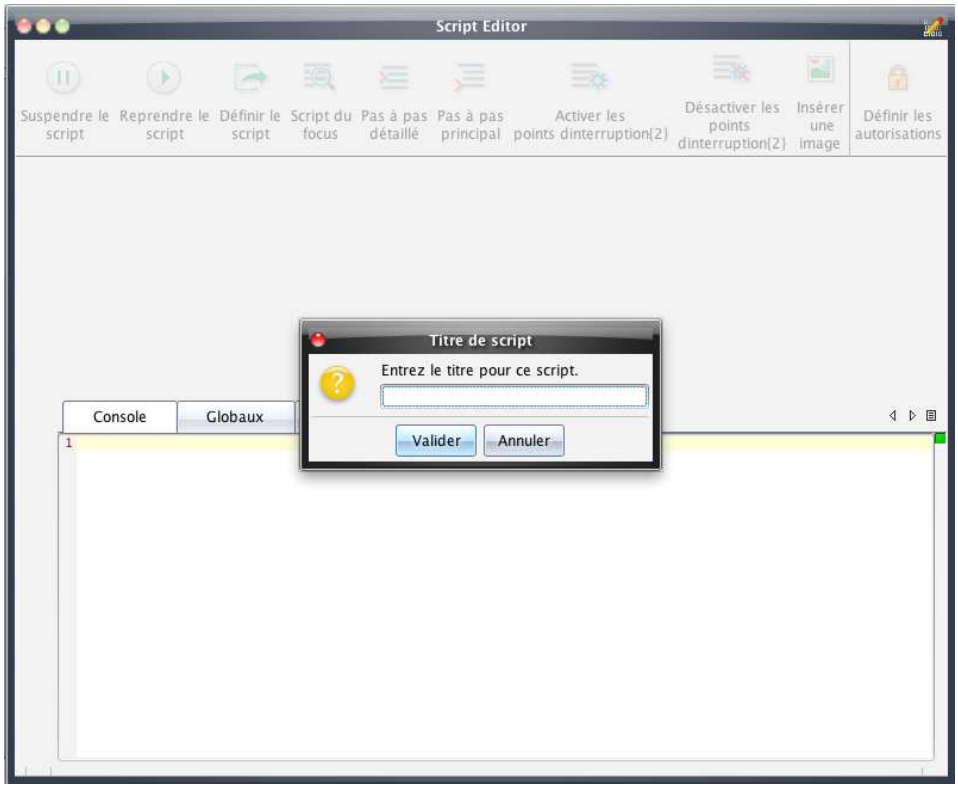
2. L'A.P.I. TI-Nspire

2.1 *L'éditeur de script*

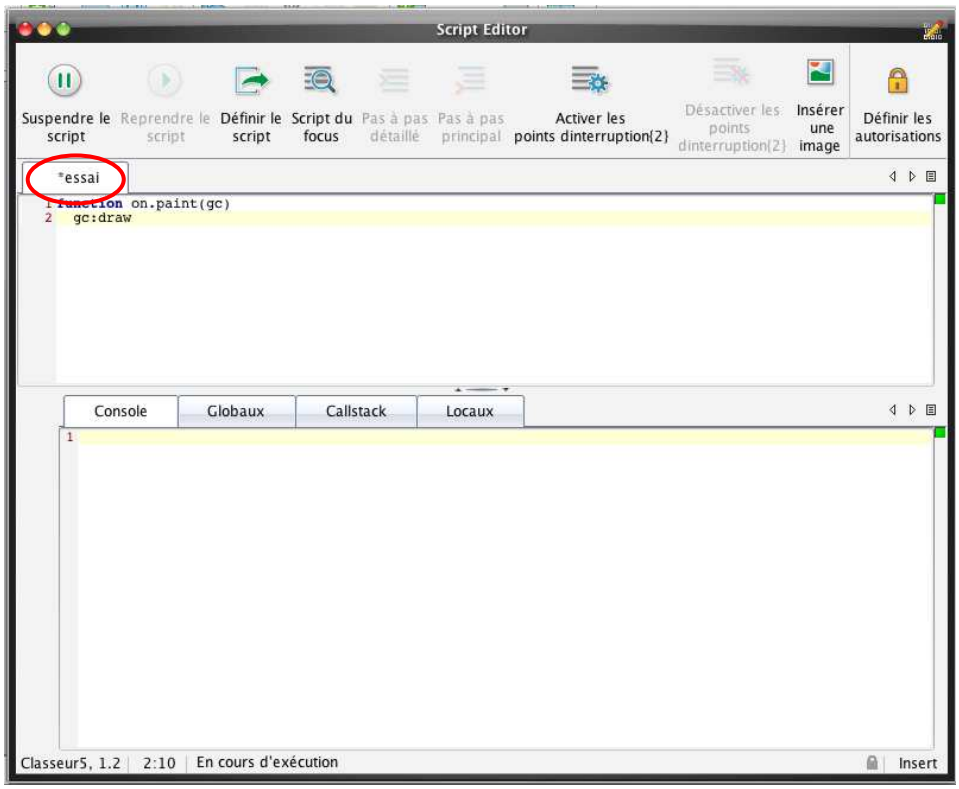
Pour utiliser l'éditeur de script il faut ouvrir un classeur, sélectionner une application (Calculs ou Éditeur mathématique par exemple), cliquer sur le menu **Insérer** : choisir **Éditeur de script** puis **Insérer le script.**



L'éditeur s'ouvre avec une fenêtre vous demandant d'entrer le titre du script.



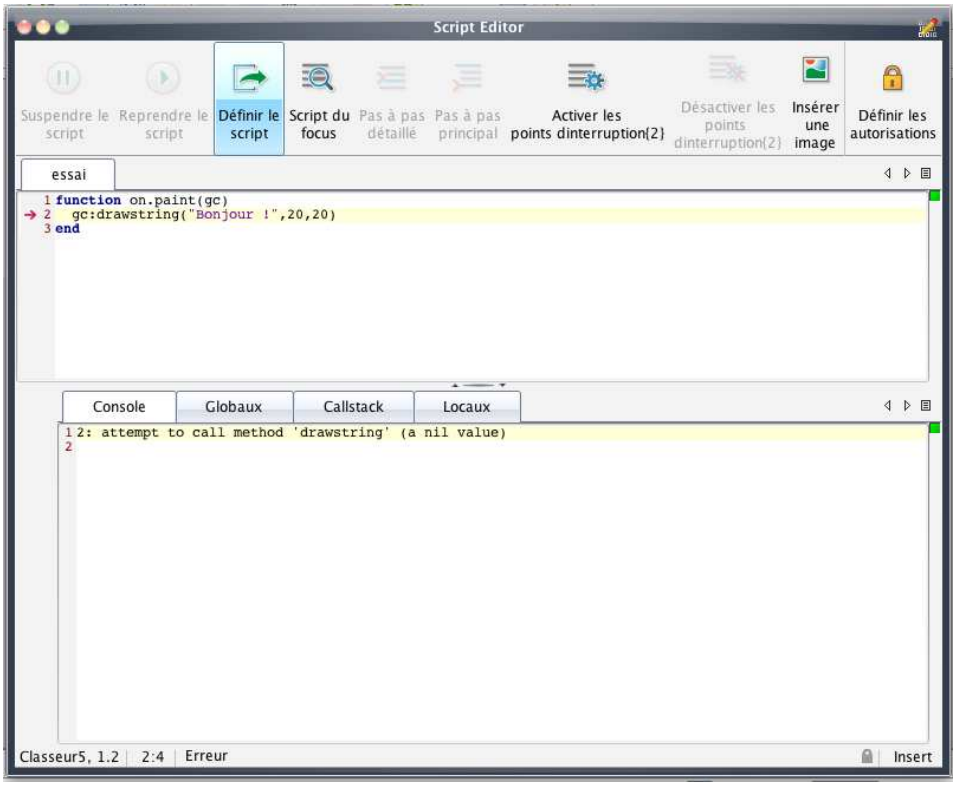
Le script étant nommé, vous pouvez commencer la saisie. Le nom apparaît en haut à gauche sous le bouton Suspendre le script précédé d'une étoile pour indiquer qu'il a été modifié et non encore sauvegardé.



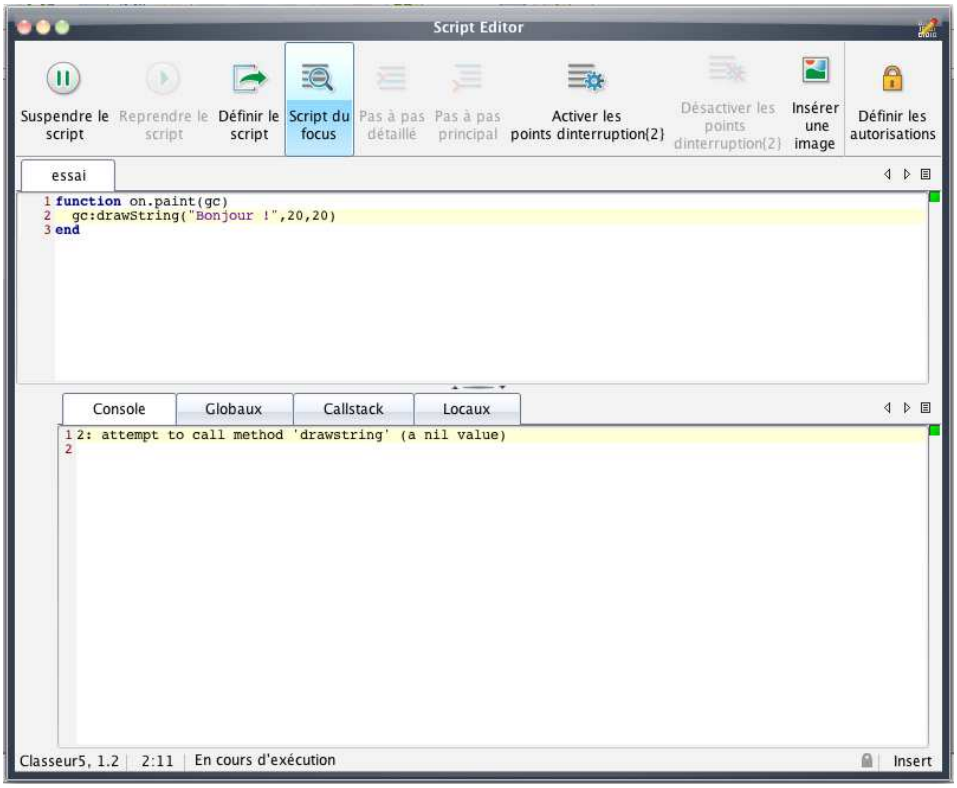
Le script une fois tapé, appuyer sur Définir le script pour le sauver dans le classeur.

Les erreurs dans le script sont signalées, vous pouvez apporter les corrections.

Dans la barre d'outils on remarquera des outils de débogage.

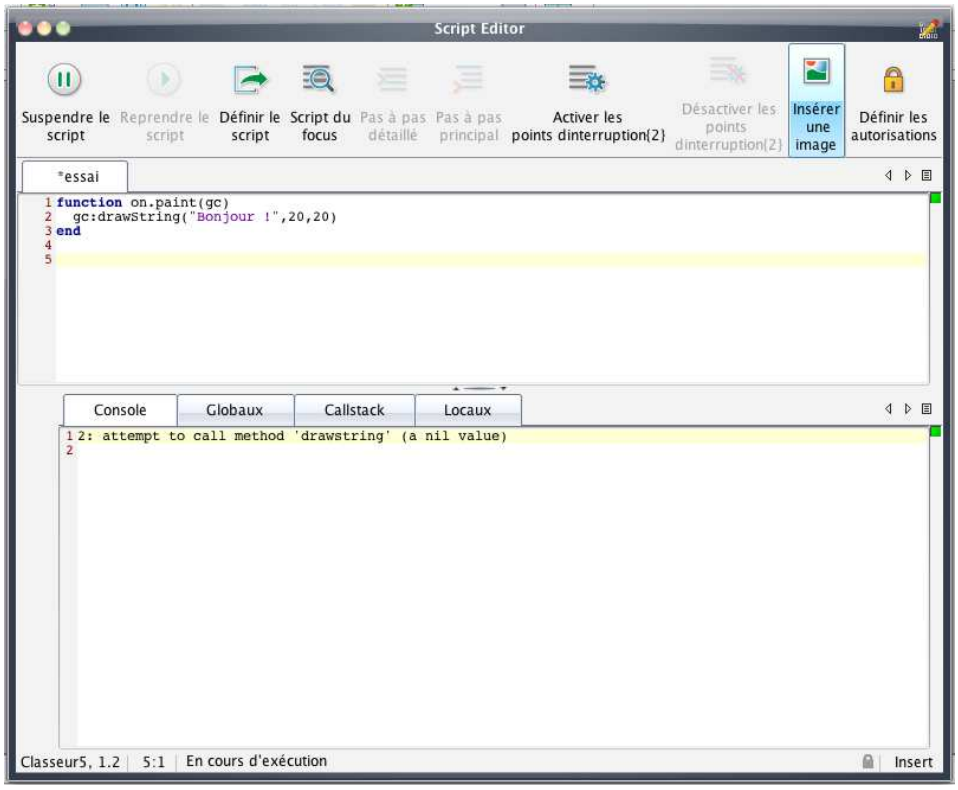


Pour voir le résultat il ne reste plus qu'à appuyer sur Script du focus (???... la traduction est surement à revoir) qui nous envoie dans le classeur attaché au script.

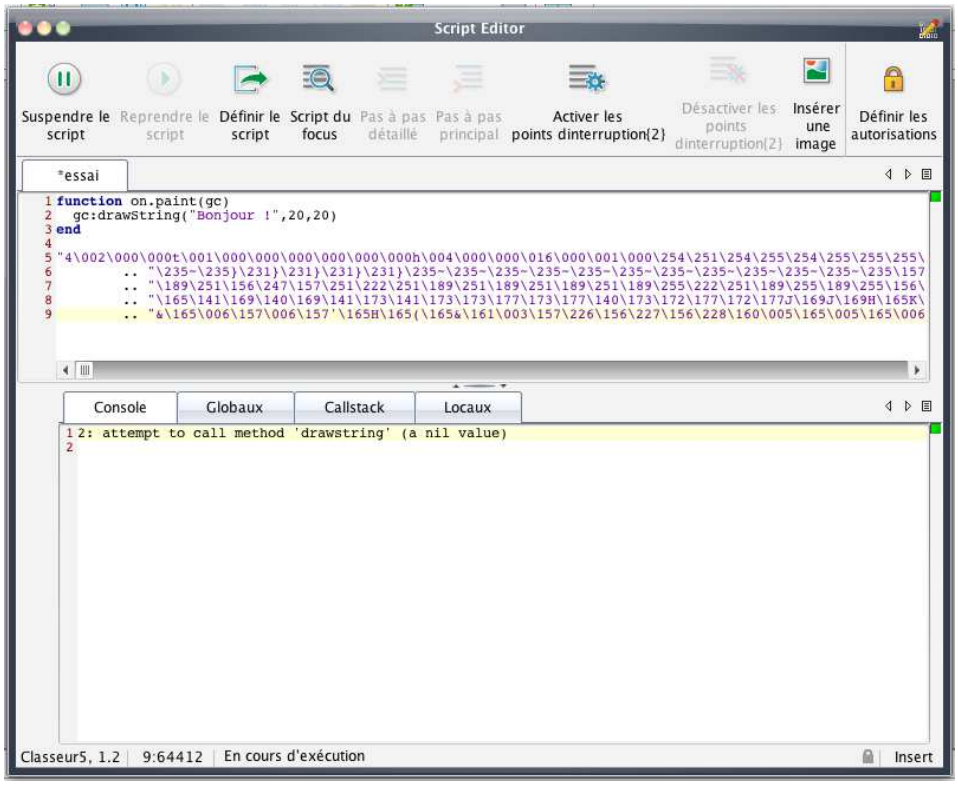


Et voici le résultat ! (modeste…)

Pour insérer une image dans un script, il faut la numériser pour cela il suffit de cliquer sur Insérer une image, un explorateur de fichier s'ouvre vous permettant de choisir le fichier correspondant à l'image à insérer. Les formats classiques sont supportés.



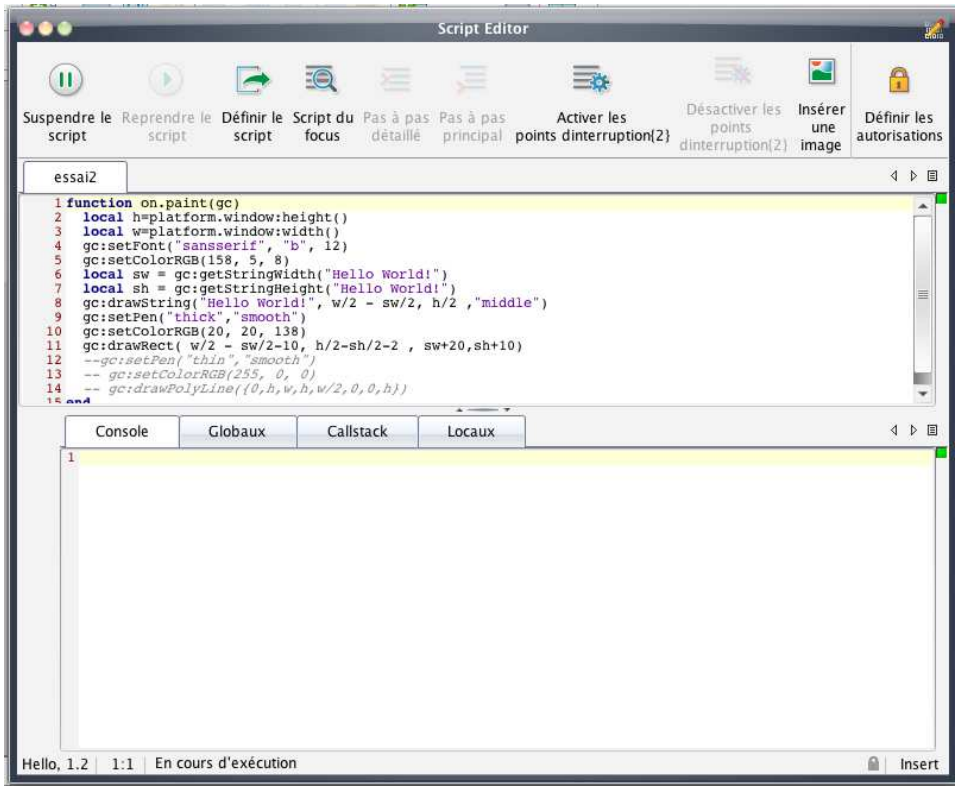
Après validation le code est collé dans le script. Voir en page 33 comment ensuite définir l'image et l'utiliser dans le script.



Le dernier bouton à droite permet de définir les autorisations. On peut protéger le script en écriture, le munir d'un mot de passe…

On peut aussi (s'il n'est pas protégé…) ouvrir le script attaché à un classeur. Il faut que la page du classeur contenant le script soit active, on clique sur **Insérer** : choisir **Éditeur de script**, là l'option **Éditer le script** n'est plus en grisé, on peut donc la sélectionner.

L'éditeur de script s'ouvre avec le script attaché au classeur affiché.



2.2 *Utilisation de l'API TI-Nspire*

Dans le premier paragraphe nous avons vu de la programmation séquentielle de Lua, mais Lua supporte également la programmation fondée sur des événements. Quand un événement se produit Lua va effectuer une instruction.

L'API possède des fonctions graphiques et de gestion de la fenêtre correspondante au script.

Il est possible d'afficher du texte, des dessins et des images.

☞ *La présentation des scripts est modifiée par rapport à la première partie. Il n'y a plus les prompts pour « coller » avec ce que l'on voit sur l'éditeur et de plus, il n'y a plus à faire la distinction entre entrée et résultat comme dans la première partie de ce chapitre.*

Afficher un texte ou un dessin

Lorsque l'on veut afficher quelque chose à l'écran (texte, image, dessin), on fait appel à la fonction **on.paint** avec l'argument gc (contexte graphique). On peut alors utiliser des fonctions graphiques fournies par l'API, comme celles-ci :

- gc:drawLine(*x1*, *y1*, *x2*, *y2*) : Dessine un segment entre les points (*x1*, *y1*) et (*x2*, *y2*)
- gc:drawPolyLine($\{x1, y1, x2, y2, \ldots, xn, yn\}$) Dessine une ligne polygonale reliant les points (*xi*, *yi*).
- gc:drawRect(*x*, *y*, *a*, *b*) Dessine au point (*x*, *y*) un rectangle de largeur *a* et de longeur *b*
- gc:drawString(*chaîne*, *x*, *y*, *pos*) Dessine au point (*x*, *y*) la chaîne de caractères, *pos* donne l'ancrage et peut avoir pour valeur "bottom", "middle", ou "top".

On peut aussi tracer des arcs et donc des cercles avec gc:drawArc et des figures pleines avec gc:fillArc, gc:fillPolygon et gc:fillRect. Les paramètres étant les mêmes que pour les fonctions draw correspondantes.

À noter également des fonctions permettant de fixer la couleur, ou la police utilisée, ainsi que l'épaisseur du trait ou sa nature (plein, pointillé, tireté).

- gc:setColorRGB(*red*, *green*, *blue*) valeurs RGB: entiers compris entre 0 et 255.
- gc:setFont(*font*, *type*, *size*), avec *font* : {"sansserif", "serif", ..}, *type* {"b", "r", "i"}, *size* (entier)
- gc:setPen(*s*, *m*) *s* {"thin", "medium", "thick"}, smooth {"smooth", "dotted", "dashed"}

Exemple :

```
function on.paint(gc)
  gc:setFont("sansserif", "b", 12)
  gc:setColorRGB(0, 70, 124)
  gc:drawString("Hello World!", 20, 20, "top")
end
```

On voudrait centrer le texte par rapport à l'écran, on a besoin pour cela de quatre paramètres, la largeur et la longueur du texte ainsi que les dimensions de l'écran. On les obtient respectivement à l'aide des fonctions : gc:getStringHeight(*string*), gc:getStringWidht(*string*) et platform.window :height, platform.window :widht. Ce qui donne :

```
function on.paint(gc)
  local h = platform.window:height()
  local w = platform.window:width()
  local sh = gc:getStringHeight("Hello World!")
  local sw= gc:getStringWidht("Hello World!")
```

```
  gc:setFont("sansserif", "b", 12)
  gc:setColorRGB(0, 70, 124)
  gc:drawString("Hello World!", w/2-sw/2, h/2, "top")
end
```

On peut encadrer le texte en rouge par exemple, il suffit de rajouter avant le **end** final :

```
  gc:setPen("thick", "smooth")
  gc:setColorRGB(255, 0, 0)
  gc:drawRect( w/2-sw/2-10, h/2-sh/2-2, sh+10)
```

Événements et boucle principale

L'API de la TI-Nspire gère les événements, nous venons de voir **on.paint** qui est appelé lorsque l'écran est marqué invalide et doit être rafraîchi. Dans l'exemple suivant, c'est l'appui sur une touche correspondant à un caractère qui va provoquer une action, on utilise pour cela **on.charIn**. Les fonctions que l'on écrit pour gérer ces événements sont prises dans une boucle principale. Dans l'exemple suivant, au départ l'écran est vide, il le reste tant que l'on n'a pas appuyé sur une touche. Si on appuie sur une touche, **on.charIn** récupère le caractère et le « colle » à la chaîne vide puis appelle platform.window:invalidate() qui marque l'écran invalide et donc à redessiner à la proche boucle à l'aide de **on.paint**.

Ce petit script écrit à l'écran les lettres que l'on tape sur le clavier en les collant les unes aux autres. Un petit gadget : la couleur des caractères change de façon aléatoire lors de la frappe.

```
input = ""                          -- initialise la variable qui va recevoir le message
function on.paint(gc)
  gc:setColorRGB(math.random(0,255), math.random(0,255), math.random(0,255))
-- on change la couleur de façon aléatoire…
  gc:drawString(input,5,5,"top")    -- dessine à l' écran le caractère frappé
end
function on.charIn(char)
  input = input..char               -- concatène le caractère à la fin du message
  platform.window:invalidate()      -- marque l' écran invalide pour qu' il soit rafraîchi
 end
```

On pourrait rajouter l'effacement du dernier caractère avec l'évement on.backspaceKey() qui est appelé lorsque cette touche est frappée, il suffit de mettre dans la fonction :

```
input = string.sub(input,0,-2)      -- voir page 14
```

Ne pas oublier après platform.window:invalidate() sinon l'écran ne sera pas rafraîchi et rien ne se passera.

Utilisation d'un timer

Comme événement déclencheur d'une action on peut aussi utilise un timer.

Dans le script suivant on crée une petite animation : une balle rouge est placée en haut de l'écran à gauche, descend lorsqu'on appuie sur une touche et s'arrête si on appuie de nouveau sur une touche.

timer.start(0.1) lance on.timer toute les 0.1 seconde, qui oblige l'écran a se rafraîchir. L'utilisation de on.charIn a été vue dans l'exemple précédent.

```
y = 5
animation = false
function on.paint(gc)
```

```
  gc:setColorRGB(255, 0, 0)
  gc: fillArc(5 − 10/2, y − 10/2, 10, 10, 0, 360)   −− tracé du disque
  if animation then
     y = y + 1                    −− incrémente la variable y
     timer.start(0.1)             −− lance l' événement on.timer toutes les 0.1 seconde
  end
end
function on.timer()
  platform.window:invalidate()    −− marque l' écran invalide pour qu' il soit rafraîchi
end

function on.charIn(ch)
  animation = not animation       −− bascule animation – arrêt animation
  platform.window:invalidate()    −− marque l' écran invalide pour qu' il soit rafraîchi
end
```

On pourrait rajouter les lignes suivantes pour ramener la balle à sa position initiale en appuyant sur la touche [enter].

```
function on.enterKey()
  y = 5
  platform.window:invalidate()
end
```

Liaison avec des variables TI-Nspire

Pour montrer la parfaite symbiose entre Lua et le logiciel TI-Nspire nous allons utiliser dans un script une variable qui pourra être pilotée à partir de l'application Calculs ou de l'application Géométrie de la TI-Nspire CAS. On va faire cela avec la fonction factorielle version récursive vue dans la première partie de ce chapitre.

Voici les fonctions disponibles pour travailler sur des variables TI-Nspire :

- var.list() – retourne la liste de toutes les variables de l'activité
- var.monitor() – permet de surveiller une variable définie dans un classeur TI-Nspire
- var.recall(string) – retourne la valeur de la variable passée en argument
- var.recallstr(string) – idem mais en convertissant le résultat en chaîne de caractères
- var.store(string, val) – stocke la valeur *val* dans la variable *string*
- var.unmonitor() – arrête la surveillance de la variable

On utilise deux événements **on.paint**(), (que nous avons déjà vu) pour « dessiner » à l'écran le résultat et **on.varChange**() événement déclenché lorsque la variable surveillée par **var.monitor**() change.

Quand la variable change, on demande la réactualisation de l'écran à l'aide de la fonction **platform.window.invalidate**(), qui passée sans paramètre, réactualise tout l'écran, sinon on peut indiquer seulement une partie à réactualiser.

Comme l'écran doit être réactualisé à la prochaine boucle du processus, **on.paint**() est appelé pour redessiner l'écran. Il ne reste plus qu'à définir les instructions permettant d'afficher le résultat.

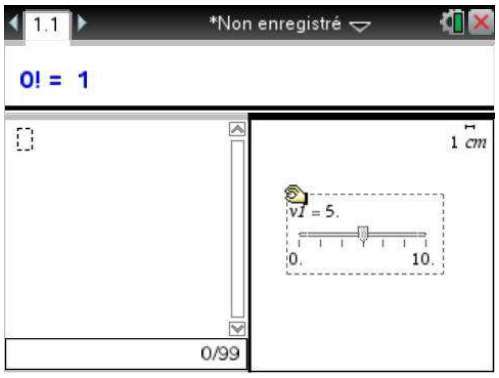
Voici le script :

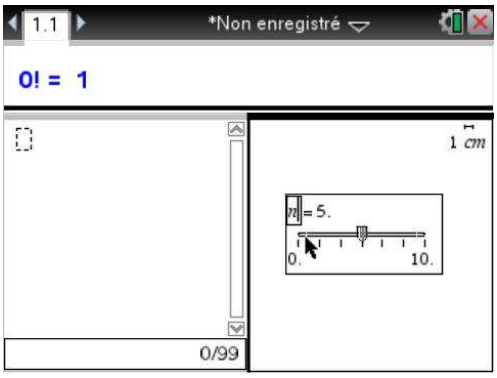
```
-- On indique ici qu'il faut surveiller le contenu de la variable n

var.monitor("n")
```

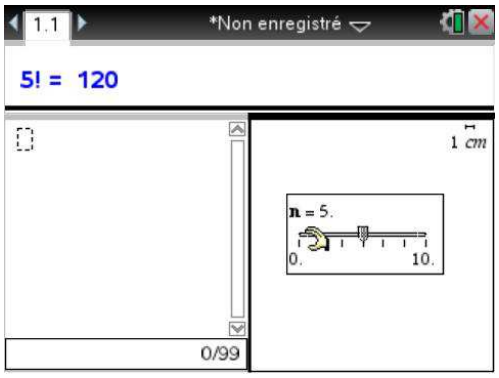
```
-- définition de la fonction factorielle (récursive)
function factorielle(n)
if math.floor(n)<n or n<0 then return "n n'est pas un entier naturel"
elseif n==0 then return 1
else return n*factorielle(n-1)
end
end
-- Quand la valeur de la variable surveillée change, l'écran est marqué invalide
-- ce qui provoque la réactualisation de son contenu à la boucle suivante
function on.varChange(n)
  platform.window:invalidate()
end
-- La fonction suivante est la fonction d'affichage l'événement provoquant son exécution
-- est déclenché par la commande platform.window:invalidate()
-- qui est exécutée chaque fois que la variable n change
function on.paint(gc)
  local wh,ww,n
  wh=platform.window:height()
  ww=platform.window:width()
  gc:setPen("medium", "smooth")
  gc:drawLine(0, 40, ww, 40)                    -- affichage d'un trait de séparation
  n=var.recall("n") or 0
  if math.floor(n)<n or n<0 then
    gc:setColorRGB(255, 0, 0)
    gc:setFont("sansserif" , "b", 10)
    gc:drawString("n n'est pas un ",5,5,"top")   -- affichage du message d' erreur
    gc:drawString("entier naturel ",5,20,"top")
  else
    gc:setColorRGB(0, 0, 255)
    gc:setFont("sansserif" , "b", 12)
    x=gc:drawString(n.."! = ",10,10,"top")       -- affichage de n!
    x=gc:drawString(factorielle(n),x+5,10,"top")
  end
end
```

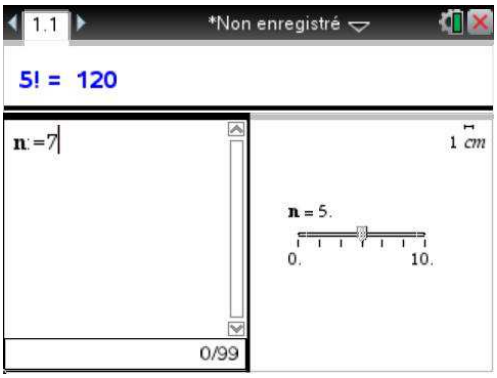
Une fois le script enregistré, on bascule sur le classeur associé à ce script. On partage la page associée au script en trois : en bas à gauche on ouvre l'application Calculs et à droite l'application Géométrie. On crée un curseur dans la partie Géométrie. On modifie la variable du curseur en lui donnant la valeur *n*. Comme on a mis 0 comme valeur par défaut 0 ! = 1 est affiché.



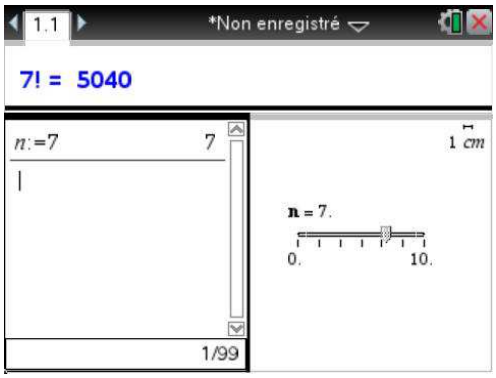


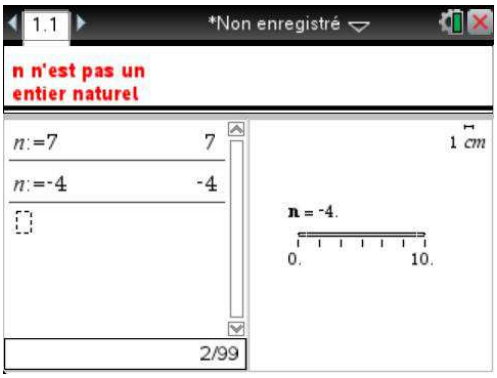
On peut alors modifier *n* soit avec le curseur (à gauche), soit dans l'application Calculs, en affectant une valeur directement à *n*.





On obtient le message d'erreur si *n* reçoit une valeur négative par exemple.





Insertion et modification d'image

La TI-Nspire permet déjà l'insertion d'image, mais il n'est pas possible de travailler dessus, par exemple de la redimensionner ou de la déplacer dans l'écran.

Pour insérer une image il faut tout d'abord la digitaliser, la commande en haut à droite de l'éditeur permet de le faire.

image.new permet de définir une image à partir de son code (pour des informations sur le codage des images TI voir : **http://wiki.inspired-lua.org/TI.Image**).

image.copy(*image*, *scx*, *scy*) retourne la copie d'une image redimensionnée, pour garder les mêmes proportions, on utilisera la commande suivante qui garde le rapport largeur/hauteur constant.

```
image.copy(image, sc * image.width(image), sc * image.height(image))
```

image.width et image.height donnent respectivement la largeur et la hauteur de l'image passée en argument.

on.resize est appelé lorsque les dimensions de la fenêtre sont modifiées.

Le script suivant définit une image, l'affiche et on peut faire un zoom avant à l'aide de la flèche vers le haut, un zoom arrière avec la flèche vers le bas (attention de ne pas faire disparaitre l'image...).

```
Im = image.new("\170\000\000\000\208\000\000\000\000\000T\016\… \156\134\156\134\")

function on.resize()
  W = platform.window:width()
  H = platform.window:height()
  Sc = 1
end

function on.arrowUp()
  Sc = Sc + 0.1
  platform.window:invalidate()
end

function on.arrowDown()
  Sc = Sc − 0.1
  platform.window:invalidate()
end

function on.paint(gc)
  local imw = image.width(Im)
  local imh = image.height(Im)
  local im = image.copy(Im, Sc * imw, Sc * imh)
  local imw = image.width(im)
  local imh = image.height(im)
  gc:drawImage(im, (W − imw)/2, (H − imh)/2)
end

function on.enterKey()
  Sc = 1
  platform.window:invalidate()
end
```

Classes

Dans l'API TI-Nspire il existe une fonction **class** qui définit une méthode permettant de créer plus facilement des classes. Ceci donnerait avec l'exemple du paragraphe 1.9.

```
Compte = class()
function Compte:new(solde)        -- constructeur de la classe
self.solde = solde or 0
end
function Compte:retrait(r)
self.solde = solde − r
end
function Compte:depot (d)
>> self.solde = self.solde + d
```

```
>> end
```

Les appels :

```
monCompte=Compte(500)          -- création d' un compte
monCompte:retrait(100)         -- on fait un retrait sur le compte
```

Pour créer une classe fille, il suffit d'écrire :

```
CompteSpecial = class(Compte)
```

pour que la nouvelle classe hérite des méthodes de la classe Compte.

Nous n'avons vu qu'un tout petit échantillon des possibilités offertes par le logiciel TI-Nspire et Lua, à vous de jouer maintenant et de laisser libre court à votre imagination !...