Stoicos

Table des matières

Concepts principaux

Intro

Le Stoicos est un langage impératif interprété en ruby dont la conception débute le 22/09/2015.

Le terme Stoicos vient du désir de créer un langage qui resterait stoïque face aux erreurs de programmation, soit en tentant de les éviter, soit en ne les prenant pas en compte.

Syntaxe

Structure

Le Stoicos est basé sur un principe syntaxique simple et unique : toute instruction est une fonction donc les arguments sont soit des arguments simples soit des arguments composites. Chaque élément est séparé des autres par une espace.

Un argument simple est un argument comme un entier, une chaîne de caractères qui peut être utilisé directement par la fonction.

Exemple :

```
print 'bonjour le monde'
allocate_var ma_variable 42
```

Dans le premier exemple on affiche « bonjour le monde ».

Dans le second on alloue à la variable **ma_variable** la valeur **42**

Un argument composite est un argument composé d'une fonction et de ses propres arguments, le tout entouré par des parenthèses, qui sera calculé avant son utilisation.

Exemple :

```
print (read_var ma_variable)
allocate_var (repeat_s c 4) (repeat_s c 4)
```

Les personnes ayant fait un peu de programmation pourront noter une subtilité dans ce dernier exemple. En effet comme signalé plus haut toute instruction est fonction ce qui signifie que dans le cadre de l'allocation de valeur à une variable, le nom de la variable est aussi un argument et peut dont être le retour d'une autre fonction. Ainsi l'exemple **allocate_var (repeat_s c 4) ((repeat_s c 4))** alloue à la variable **cccc** la valeur **cccc**.

Omission ou surcharge d'arguments

En Stoicos il est possible de donner à une fonction plus d'arguments que nécessaire ou d'en omettre.

Si la surcharge d'arguments n'aura aucun effet (les arguments supplémentaires seront simplement ignorés) le cas de l'omission peut se révéler utile dans quelques rares cas où les arguments seront remplacés par des arguments par défaut.

Ces deux possibilités n'ont cependant qu'une maigre utilité et servent surtout, en cas d'erreur, à éviter un plantage malencontreux du programme.

Décomposition d'une instruction

Le langage étant évalué ligne par ligne, on pourrait se retrouver avec des lignes très longues et illisibles. De fait il est possible de séparer une ligne en plusieurs sous-lignes en recpectant la syntaxe suivante :

```
Sous-ligne 1
      Sous-ligne 2
      Sous-ligne 3
      Sous-ligne 4
      Sous-ligne 5
      Sous-ligne 6
```

On pourra noter que les tabulations utilisées ici peuvent être remplacées par des espaces (tant qu'on en met au moins un) et que le nombre de tabulation ou d'espace est laissé libre au choix de son utilisateur. On pourrait donc écrire :

```
Sous-ligne 1
      Sous-ligne 2
            Sous-ligne 3
      Sous-ligne 4
        Sous-ligne 5
    Sous-ligne 6
```

Cependant, un code étant préférable lorsqu'il est lisible, il vaux mieux ne l'utiliser que lorsque cela apporte du sens au code.

Commentaires

Il est possible de mettre du code en commentaire pour que celui-ci ne soit pas interprété par le langage. Deux méthodes sont possible pour mettre du code en commentaire :

Tout d'abord on peut commencer une ligne par un « # », ce qui aura pour effet de la commenter, elle et toutes les sous-lignes qui la suivent. Ce qui signifie qu'un bloc de plusieurs sous-lignes pourra être mis en commentaire en ajoutant un « # » simplement à la première ligne.

Ensuite il est possible d'entourer plusieurs instructions en plaçant un « =begin » dans une ligne avant et un « =end » dans une ligne après. A noter que les lignes ne devront pas contenir un caractère de plus pour être prises en compte par l'interpréteur.

Quelques exemples de code

Exemples simples

Dans cette partie tutoriel, nous allons voir quelques exemples de code pour pouvoir se familiariser avec la syntaxe du Stoicos. Pour commencer, rien de plus logique que de présenter le traditionnel « bonjour le monde » :

```
#Affiche bonjour le monde
print 'bonjour le monde'
```

Et d'autres exemple simples qui affichent une donnée stockée dans une variable :

```
allocate_var reponse 42
#Affiche 42
print (read_var reponse)
#Affiche j ai 42 ans
print (format 'j ai {} ans' (read_var reponse))
```

Bien sûr, un code se voulant interactif, on peut demander une entrée de l'utilisateur via la commande « input » :

```
#Récupère l'input
allocate_var age (input 'quel est ton age ? : ')
#Affiche le résultat
print (format 'ah tu as donc {} ans !' (read_var age))
```

Et si on voulais considérer qu'une personne est veille à partir de 25 ans, sinon jeune :

```
#Récupère l'input
allocate_var age (input 'quel est ton age ? : ')
#Affiche le résultat selon l'input
print (if (supeq (read_var age) 25)
      (format 'tu as {} ans donc tu es vieux !' (read_var age))
      (format 'tu as {} ans donc tu es jeune !' (read_var age))
      )
```

Maintenant si on souhaite afficher tous les entiers de 0 à n

```
#Récupère l'input
allocate_var n (input 'compter jusqu ou ? : ')
#Compte jusqu'à n
each (rangex (read_var n)) i (print (read_var i))
```

Si on veut calculer le factoriel d'un nombre n

```
#Récupère l'input
allocate_var n (input 'Factoriel de : ')
#Affiche le factoriel s'il s'agit d'un entier positif, ou sinon une erreur
print (if (inf (read_var n) 0)
      'les nombres negatifs n ont pas de factoriel'
      (let_in temp
            (prod_arr (rangex 1 (read_var n)))
            (format 'le factoriel de {} vaut {}' (arr (read_var n) (read_var temp)))
            )
      )
```

Liste des fonctions

Afin de mieux comprendre les fonctions veuillez notez l'utilisation des syntaxes suivantes :

*Arguments signifie qu'autant d'arguments que voulu peuvent être mis.

[Argument] signifie que l'argument peut être omis.

Fonctions générales

print valeur

Affiche une valeur (convertie en string) et la retourne.

input [string]

Affiche string (si renseigné) et récupère sous forme d'un string une entrée de l'utilisateur.

Manipulation de variables

En stoicos les variables sont globales. De plus une variable non ititialisée dans le programme (aucune valeur ne lui a été allouée par la programmeur) à par défaut la valeur " (une chaîne de caractères vide).

allocate_var nom contenu

Alloue à la variable nom la valeur contenu

read_var nom

Retourne le contenu de la variable nom

swap_var nom nombis

Donne à la variable nom la valeur de la variable nombis et vice versa

let_in nom contenu contexte

Permet de créer une variable pseudo-locale nom. Lui sera allouée la valeur de contenu avant de d’exécuter un contexte supposé l'utiliser. Par exemple

```
print (let_in temp
      (pow 10 2)
      (rangex 1 (read_var temp))
      )
```

L'intérêt principal de let_in est de pouvoir diviser une combinaison d'instructions complexe en deux plus petites. De plus cela évite de surcharger le programme avec trop de variables globales. En effet, à la fin de l’exécution de contexte la variable nom est supprimée.

Manipulation de chaînes de caractère

Par défaut en Stoicos, tout est chaîne de caractères. Ce qui signifie qu'il n'y à pas besoin d'utiliser de ' ou de " pour définir une chaîne. Cependant l'espace étant considérée comme un séparateur d'arguments plutôt qu'un caractère il faut entourer les chaînes de caractères l'utilisant.

Par exemple :

```
'Ceci est une chaîne'
ceci_l_est_aussi
chaine
```

Voici la liste de fonctions de manipulation de chaînes de caractères :

repeat_s str num

Retourne une chaîne de caractère comprenant nul fois la chaîne char.

capitalize str

Retourne str avec le premier caractère mis en majuscule.

upcase str

Retourne str avec tous les caractères mis en majuscule.

downcase str

Retourne str avec tous les caractères mis en minuscule.

swapcase str

Retourne str avec tous les caractères en minuscule mis en majuscule et vice versa.

Title str

Retourne str avec tous les caractères en début de mot mis en majuscule.

Reverse str

Retourne str avec les caractères mis dans l'ordre inverse.

split str char

Retourne un tableau d'éléments de str en utilisant char comme séparateur.

format str tab

Retourne une chaîne où les {} sont remplacés par les éléments de tab. Si un seul élément est dans tab, alors tab peut ne pas être un tableau.

at_s str num

Retourne un caractères d'une chaîne str à la position num (0 étant le position du premier caractère).

slice_s str num1 num2

Retourne une sous-chaîne de la chaîne str comprise entre les indexes num1 et num2 inclus. Si num1 est égal à num2, retourne une chaîne vide.

include_s str strbis

Retourne true si la chaîne str contient la sous-chaîne strbis, false sinon.

concat_s str strbis

Retourne la concaténation de str et strbis.

Manipulation d'entiers

Incr num

Retourne num + 1.

decr num

Retourne num - 1.

sum num numbis

Retourne la somme de num et numbis.

dif num numbis

Retourne la différence entre num et numbis.

prod num numbis

Retourne le produit de num et numbis.

div num numbis

Retourne la division de num par numbis.

mod num numbis

Retourne le modulo de num par numbis.

pow num numbis

Retourne num à la puissance de numbis.

Divisible num numbis

Retourne true si num est divisible par numbis, false sinon.

Even num

Retourne true si num est paire, false sinon.

Odd num

Retourne true si num est impaire, false sinon.

Manipulation de booléen

and element elementbis

Retourne true si les deux éléments sont true, false sinon.

or element elementbis

Retourne true si au moins un élément est true, false sinon.

not element

Retourne true si l'élement est false et false s'il est true.

equal element elementbis

Retourne true si les deux éléments ont la même conversion en string, false sinon.

different element elementbis

Retourne true si les deux éléments n'ont pas la même conversion en string, false sinon.

inf int inbis

Retourne true si int est inférieur à intbis, false sinon.

sup int inbis

Retourne true si int est supérieur à intbis, false sinon.

infeq int inbis

Retourne true si int est inférieur ou égal à intbis, false sinon.

supeq int inbis

Retourne true si int est supérieur ou égal à intbis, false sinon.

true

Retourne true.

false

Retroune false.

Manipulation de liste

arr *elements

Retourne une liste elements

Par exemple, pour créer la liste [1, 2, 3] :

```
arr 1 2 3
```

rangex int [intbis]

Retourne une liste contenant les entiers de int (inclus) à intbis (inclus). Si intbis n'est pas donné, retourne les entiers de 0 (inclus) à int (inclus).

head liste

Retourne le premier élément d'une liste.

tail liste

Retourne tous les éléments d'une liste sauf le premier.

sort liste

Retourne la liste triée.

sum_arr liste

Retourne la somme des entiers contenus dans la liste.

prod_arr liste

Retourne le produit des entiers contenus dans la liste.

repeat element int

Crée une liste de int éléments.

prepend liste element

Retourne la liste avec element rajouté en première place.

append liste element

Retourne la liste avec element rajouté en dernière place.

remove liste element

Retourne la liste avec element retiré une fois.

remove_all liste element

Retourne la liste avec element retiré pour toutes ses occurences.

filter liste iterateur contexte

Parcours la liste avec iterateur et ne garde que les éléments renvoyant true quand on leur applique le contexte.

Par exemple pour conserver les nombres pairs inférieur à 10 :

```
filter (rangex 9) i (divisible (read_var i) 2)
```

map liste iterateur contexte

Parcours la liste avec iterateur et et applique à tous les éléments le contexte.

Par exemple pour mettre au carré les nombres inférieurs à 10

```
map (rangex 9) i (pow (read_var i) 2)
```

len_arr liste

Retourne la taille de la liste.

at_arr liste int

Retourne l'element de la liste situé en position int.

slice_arr liste int intbis

Retourne une sous-liste de liste contenant les éléments de la positions int (inclus) à intbis (inclus).

include_arr liste element

Retourne true si la liste contient l'element, false sinon.

concat_arr liste listebis

Retourne la concaténation de liste et listebis.

Manipulation de procedure

En Stoicos, les procédures (ou procs) sont des string contenant des fonctions qui n'ont pas encore été calculés par l'interpréteur. Leur but est d'être calculé lorsque le programmeur le souhaite, souvent bien après leur écriture, et de pouvoir être appelé à volonté sans avoir à tout réécrire.

De plus cette catégorie concerne toutes les fonctions utilisant des procédures telles que les boucles et les conditions.

do *actions

Prend une liste d'actions et les execute dans l'odre. Sert principalement comme argument de fonctions qui prennent une action en argument pour pouvoir en mettre plusieurs à la place.

Retourne la valeur de la dernière actions.

proc_w *actions

Retourne un proc à partir d'une liste d'actions.

proc_r proc

Prend un proc et l'effectue. Retourne la dernière valeur manipulée par le proc.

each liste iterateur action

Parcours une liste en stockant chaque valeur dans l'iterateur et en effectuant ensuite l'action. Retourne le résultat de la dernière action effectuée.

each_char chaine iterateur action

Même chose que each mais effectué sur une chaine de caractères.

while bool action

Effectue action tant que bool retourne true. Retourne la valeur de la dernière action une fois stoppé.

if bool action actionbis

Si bool revois true effectue et retourne action, sinon effectue et retourne actionbis.

case *(bool, action) [reste]

Parcours les paires (bool, action) jusqu'à tomber sur un bool qui revoie true. Effettue et retourne alors l'action qui lui est associée. Si reste est présent et qu'aucun bool n'a renvoyé true, reste est effectué et retourné.

pass

Retourne un string vide. Utile pour faire quelque chose qui ne fait rien.