Vuziks

Projet de théorie des langages

Guillaume DELAPRÉS & Guillaume VILLEREZ

Table des matières

Utilisation

Mode interpréteur

En mode interpréteur, le fichier passé en paramètre est lu puis interprété directement.

Mode interactif

En mode interactif, c'est à l'utilisateur d'entrer à la suite les données du programme. Dans le cas des boucles et conditions, celles-ci doivent se suivre d'une fin d'instruction pour être correctement reconnues et interprétées :

```
;
```

Options du programme

Il est possible de lancer le programme avec différentes options modifiant son comportement :

- **-h** / **--help** : Affichage de l'aide
- **-v** / **--version** : Affichage de la version du logiciel
- **-i** / **--interactive** : Version avec prompt interactif (de base si pas de fichier lié)
    - **--auto-dump** : Dump auto des données présente en fin de l'évaluation du statement
- **--show-op** : Déroule le détail des opérations après les avoir analysées, avant l'évaluation
- **--verbose** : Affiche plus d'informations lors du déroulement du programme
- **--mem-info** : Montre le poids total en mémoire du programme, et le nombre de blocs n'ayant pas été libéré
- **--show-timer** : Montre le chrono de parsage et le chrono d'exécution

Langage

Syntaxe générale du langage

La syntaxe générale du **vuziks** est très proche de celle du *langage C* : chaque unité de code doit être entourée d'accolades (A l'exception de l'unité générale de base, implicite), et chaque ligne de code doit se terminer par un point-virgule. Contrairement au *C* les commentaires sont fait avec le signe `#` (ou `<#` et `#>` pour les multi-lignes)

Exemple:

```
# Exemple
 new var a = "hello world";
 console.println(a);
```

D'autres codes d'exemple sont disponibles sur le **github** du projet à l'adresse suivante : https://github.com/Vuzi/vuziks/tree/master/exemples

Variables

Les variables en **vuziks** ne sont pas typées, et peuvent donc servir à être affectée à de multiples types de valeurs. Une variable est créé avec le mot clef `new var` (Variable temporaire) ou `new attr` (Variable membre) et prend par défaut la valeur `null`, qui ne correspond à aucune valeur.

```
 new var a = true;
 a = 2.5;
 print(a); # affiche 2.5
```

Il existe également des variables naturelles, entrées directement dans le code (Comme les nombres), à la différence des variables dynamiques qui sont créées à l'exécution.

Types gérés

- numérique
- booléen
- tableaux
- référence
- unité
- objets
- null (Non initialisé / Aucun résultat)
- type (Type de variable)
- non-existant (Type renvoyé quand on accède à une donnée n'existant pas)

Par mesure de simplicité, pour le langage et pour ceux l'utilisant, il n'existe pas de conversion implicite, ni même explicite. Le **vuziks** utilise en effet un seul type de variable numérique, correspondant au type *double* du *C* stocké sur 64bit, assurant ainsi une gestion transparente des nombres.

Il n'est pas possible d'effectuer des opérations sur plusieurs types (Qui de toute façon n'auraient pas de sens). De même, les comparaisons de valeurs (Plus grand que, etc..) ne peuvent s'effectuer que sur des types numériques. La liste complète est disponible avec celle des différentes opérations.

Type numérique

Le type numérique gère les flottant de manière naturelle. Il n'existe pas d'entiers dans notre langage, mais ceux-ci peuvent être utilisés pour stocker des entiers sans problèmes (Sur 53 bit, ce correspond à une taille supérieure des entiers en *C* (*int* sur 32bit). Il existe donc un opérateur de division entière si l'on souhaite une division entière.

On peut écrire un numérique de plusieurs manières :

```
new var a = 12; # Sans partie flottante
new var b = 12.0; # Même valeur que a, a == b renverra vrai
```

Type booléen

Le type booléen ne peut prendre que les valeurs **true** et **false**. Il s'agit du résultat de chaque opération logique. On peut les utiliser dans des variables avec les valeurs **true** et **false**.

```
new var a = true; # a vaudra vrai
```

Type objet

Un *vuziks* un *objet* n'est pas un objet au sens orienté objet du terme. En effet le langage ne gère pas des éléments comment les droits d'accès ou l'héritage. Ici l'objet représente simplement un contexte d'exécution avec les variables associées.

Les objets, tout comme les autres types de variables complexes, fonctionne par référence et non par copie.

On peut déclarer un objet avec le mot clef `new` suivit de l'appel à la fonction correspondante, qui sert alors de constructeur : la valeur du `return` (si présente) est ignorée, et les attributs définis stockés dans l'objet créé.

Type chaîne de caractère

Les chaines de caractère sont des données qui permettent de stocker des caractères. Pour créer une chaîne de caractère, il faut utiliser les guillemets double `"` ou simple `'` et écrire sa chaîne entre. Il est possible d'effectuer des opérations sur les chaines, comme par exemple des concaténations.

```
new var a = "Hello";
new var b = " world!";

console.println(a.append(b)); # Affiche Hello world!
```

Type tableau

Les tableaux sont un type de donnée qui permet de stocker des variables génériques en utilisant un système de clef. Pour créer un tableau, il convient d'utiliser les crochets **[** et **]** comme ceci :

```
new var a = [12, 14, 16]; # Tableau classique
new var b = []; # Tableau vide
new var c = ['a' => 12, 'b' => 14, 'c' => 16]; # Tableau associatif
```

Pour le moment les clefs des tableaux sont uniquement des nombres entiers. Il est possible d'ajouter des valeurs à l'aide la fonction *add,* d'en récupérer avec les crochets ou *get,* d'en avoir la taille avec *lentgh,* et d'en récupérer le dernier élément avec *pop*.

Type fonction

Une fonction désigne en réalité simplement un lien vers une unité de code du langage. Il n'existe pas de liste de fonctions/classes dans le langage, c'est pourquoi il est nécessaire de stocker les références de ces unités de traitement dans des variables.

D'un point de vue fonctionnement, tout le code est analysé avant traitement, signifiant que l'unité à déjà été traitée et stockée en mémoire. Chaque variable contient en réalité uniquement une référence vers cet élément, permettant de construire un objet ou d'effectuer une opération suivant si on sauvegarde ou non son contexte d'exécution :

```
new var ref = fonction(a) {
   return a + 1;
};

// Utilisé comme une fonction
new var r = ref(12);
```

Type 'type'

Ce type représente un type de variable. Il est possible de récupérer le type d'une variable avec l'opérateur `typeof`, et de comparer le type d'une variable avec l'opérateur `is` :

```
typeof a; # type de a
a is num; # true si a est un nombre, faux sinon
```

Type null

Ce type représente soit une variable non initialisé, soit une valeur de retour d'une fonction ne renvoyant aucun résultat. On peut affecter cette valeur à un nombre avec le mot clef `null`. Il est important de tester cette valeur avec `===` (Opérateur de comparaison de type), sinon celui-ci testera s'il s'agit des même `null` (Stocké à la même adresse mémoire).

```
new var a; # a vaut null
new var b; # b vaut null

a == b; #  vaut false
a === b; # vaut true
```

Type non-existent

Ce type représente une variable n'existant pas en mémoire. Il n'est logiquement impossible d'affecter une valeur de ce type dans une variable. Tout comme le type `null`, il est important de tester avec `===` plutôt qu'avec un simple `==`.

Cette valeur peut servir de test, par exemple pour un accès tableau :

```
new var d = [1, 2, 3];
if(d[5] === nonexistent) {
   # La case n'existe pas
} else {
   # La case existe
}
```

Il est également possible d'utiliser l'opérateur de test d'existence `?` qui permet de vérifier plus simplement si une valeur existe ou non :

```
if( ?a ) # si a existe
    return a;
else # sinon..
    return 0;
```

Durée de vie des variables

Les variables naturelles sont toujours passées en copie, alors que les variables complexes (Contenant autre chose que des variables naturelle : tableau, objet) sont passées en référence. Si l'on souhaite effectuer une copie, il faudra la réaliser manuellement.

Variables temporaires

Ce type de variable, déclarée avec `new var` a pour durée de vie l'exécution du bloc. Une fois cette exécution terminée, elles ne seront plus accessibles (Par exemple dans le cas d'une condition ou d'une fonction).

Variable membres

Ce type de variable, déclarée avec `new attr` a pour durée de vie celle de l'objet où elle est associée. Ils sont accessible avec l'opération `.` à côté de l'objet en question. Toute fonction appelée à l'intérieur d'un objet, même depuis l'extérieur, pourra utiliser les attributs de l'objet ou elle est stockée.

Opérateurs

Il existe des opérateurs de calcul, permettant d'effectuer des opérations mathématiques entre deux nombres, et des opérateurs de comparaisons qui permettent de comparer des nombres/tableau/objets entre eux, ainsi que des opérateurs logiques et spéciaux.

Calcul

Signe	Définition	Type attendu
`a + b`	Addition de deux nombres	Numérique
`+a`	Plus unaire	Numérique
`a - b`	Soustraction de deux nombres	Numérique
`-a`	Moins unaire	Numérique
`a * b`	Multiplication de deux nombres	Numérique
`a / b`	Division de deux nombres	Numérique
`a : b`	Division entière de deux nombres	Numérique
`a % b`	Modulo de deux nombres	Numérique
`a ^ b`	Puissance d'un nombre par un autre	Numérique
`a = b`	Affectation (Uniquement possible vers une variable dynamique)	Tout type

Comparaison

La particularité de ces opérateurs est de donner comme résultat un booléen. Certains attendent même un booléen en entrée.

Signe	Définition	Type attendu
`a > b`	Supérieur	Numérique
`a >= b`	Supérieur ou égal	Numérique
`a < b`	Inférieur	Numérique
`a <= b`	Inférieur ou égal	Numérique
`a == b`	Égalité	Tout type
`a != b`	Différence	Tout type
`a === b`	Comparateur de type	Tout type
`!a`	Contraire logique	Booléen
`a and b`	ET logique	Booléen
`a or b`	OU logique	Booléen
`?a`	Test d'existence	Tout type

Autre

Signe	Définition	Type attendu
`a.b`	Accession d'attribut	Objet
`var a`	Recherche de a dans les variables	Tout type
`attr a`	Recherche de a dans les attributs	Tout type
`new var a`	Nouvelle variable a	-
`new attr a`	Nouvel attribut a	-
`function() {--};`	Nouvelle fonction	-
`new a()`	Nouvel objet en utilisant a comme constructeur	Objet
`dump a`	Affiche les informations de a	Tout type
`typeof a`	Retourne le type de a	Tout type
`a is b`	Test si le type de a correspond à b	Type
`return a`	Termine la fonction courante en retournant la valeur de a	Tout type
`break`	Termine la boucle courante	Aucun

Conditions

Les conditions, ou bloc conditionnel, sont des instructions de code qui ne sont exécutée que si la condition présente est validée (Que son résultat, vaut vrai).

Si le résultat est un nombre, toute valeur autre que 0 sera considérée comme vraie. Si c'est une variable complexe, alors le résultat sera toujours vrai. Enfin, s'il est `null` ou `non-existent` alors il sera considéré comme faux.

Une condition est soit suivie d'un bloc de donnée délimité par des accolades `{ -- }`, soit directement par une seule instruction.

if (si)

La condition la plus simple est sans doute le `if`. Si la condition est vraie, alors le bloc est exécuté :

```
var val = 12;
# Affiche 42 uniquement si foo renvoi 23 avec 12 comme argument
if(foo(val) == 23) {
   print(42);
}
```

else if (sinon si)

Cette condition ne peut prendre place qu'après un premier `if` et teste une condition après que celle précédente ait été fausse. Plusieurs `else if` peuvent être situé à la suite :

```
var val = 12;
# Affiche 42 uniquement si foo renvoi 23 avec 12 comme argument
# ou alors affiche 12 si foo ne renvoi pas 23 mais que boo renvoi 12
if(foo(val) == 23) {
   print(42);
} else if (boo(val) == 12)
   print(12);
```

Une suite de condition ne peut pas commencer par un `else if`.

else (sinon)

Il s'agit d'un `else if` qui serait toujours vrai. La place de ce bloc est donc généralement après une suite de `else if` ou un `if`. Une suite de condition ne peut pas commencer par un `else`.

Boucles

Une boucle est un traitement qui sera effectué plusieurs fois. Une boucle peut posséder un test d'arrêt et des opérations à chaque tour.

Boucle simple

Une boucle simple n'a pas de condition d'arrêt, elle doit être manuellement arrêtée avec un `return` ou un `break`

loop (boucle)

La boucle `loop` est la seule boucle simple disponible :

```
loop {
   a = a - 1;
   if(a == 0)
      break;
}
```

Boucles itératives

Une boucle itérative contient une condition d'arrêt, ainsi que parfois des opérations à effectuer à chaque tour de boucle (En plus des opérations du bloc).

while (tant que)

La boucle `while` permet d'exécuter un bloc de code tant que la condition testée est vraie :

```
# Programme qui va afficher 5 fois la lettre a
var a = 5;
while(a > 0) {
  a = a - 5;
  console.log("a");
}
```

for (pour x tant que)

La boucle `for` agit comme la boucle `while` à la différence que celle-ci effectue une opération au lancement et à la fin de chaque tour.

```
# Programme qui va afficher 5 fois la lettre a
for(a = 5; a > 0; a = a - 1) {
  console.log("a");
}
```

Fonctions

Une fonction est un bloc d'instruction qui n'est pas directement appelé (Contrairement aux blocs conditionnels et itératifs) mais gardé en mémoire pour être appelé plus tard avec des variables spécifiques (Arguments).

Toutes les fonctions sont anonymes, il est donc important de stocker leurs références dans des variables pour pouvoir les utiliser.

Une déclaration de ce type de bloc se fait avec le mot clef `function` suivit du nom des arguments entre parenthèses :

```
# Fonction foo qui prend en argument deux variables a et b
new var foo = function(a,b) {
   return(a+b);
};
```

L'appel à `return` coupe le traitement et renvoi le résultat (Si un résultat est présent avec le mot clef).

Un appel de fonction en **vuziks** se fait avec le nom de la variable stockant un lien vers la fonction suivit de parenthèses avec les arguments du programme.

```
print foo(12,42) ;
```

Objet

Un objet est une variable contenant d'autres variables. On déclare un objet à l'aide d'une fonction et du mot clef `new`. La fonction appelée sera exécutée et son contexte objet sera sauvegardé. Par exemple :

```
# Constructeur
new var ma_class = function(a) {
   new attr a = a;
   new attr b = a + 2;
   new var c = 42;
};

# Instanciation
new var obj = new ma_class(40);

console.log(?a.a);  # True
console.log(?a.b);  # True
console.log(?a.c);  # False
```

Il est possible de tester la présence d'un attribut avec l'opérateur d'existence `?`. Il n'est pour l'instant pas possible d'ajouter des attributs à des objets déjà existants.

Pour l'instant, on doit stocker un objet dans une valeur pour pouvoir accéder à ses attributs.