

Le langage de programmation Scriptol

Manuel de référence complet pour la version 7.0 de Scriptol.

Le langage est entièrement implémenté dans le compilateur de Scriptol en PHP ou binaire.

- L'inclusion en page Web est possible seulement avec le compilateur de Scriptol en PHP.
- Utiliser XML comme structure de données est possible avec le compilateur en binaire.
- L'interpréteur est un outil d'apprentissage et ne reconnaît qu'une partie du langage pour le moment (voir la page "Change" pour plus de détails).
- XML est traité par l'interpréteur comme une structure dynamique.
- XML est traité par le compilateur C++ comme une classe nommée "dom" et décrite dans le document "dom.html". Dans un futur proche, le format XML de l'interpréteur sera étendu aux deux compilateurs.

Mail: webmaster@scriptol.fr

Home page: <http://www.scriptol.fr/>

Node:Top, Next:[Généralités](#), Previous:([dir](#)), Up:([dir](#))

Table of Contents

- [Généralités](#)
- [Au sujet de ce manuel](#)
- [Interpréter un programme scriptol](#)
- [Compiler un programme Scriptol en PHP](#)
- [Compiler un programme Scriptol en binaire](#)
- [Mon premier programme](#)
- [Projet Scriptol](#)
- [Fichier source Scriptol](#)
- [Caractéristiques du langage](#)
- [Page HTML](#)
- [Instruction](#)
- [Commentaire](#)
- [Symboles](#)
- [Identifieurs et mot-clés](#)
- [Variables ou primitives](#)
- [Litéraux](#)
 - [Guillemets et échappement](#)
 - [Variable dans une chaîne](#)
- [Déclaration](#)
- [Constante](#)
- [Nil et null](#)
- [Assignement](#)
 - [Assignement simple](#)
 - [Assignement augmenté](#)
 - [Assignements multiples](#)
 - [Assignement conditionnel](#)
- [Opérateurs](#)

- [Expression](#)
 - [Précédence](#)
- [Fonction](#)
 - [Alias: passage par valeur ou par référence](#)
 - [Valeurs par défaut](#)
 - [Scope et fonction](#)
 - [Le mode silencieux](#)
 - [La fonction "main"](#)
- [Print et echo](#)
 - [Input](#)
- [Structures de contrôle](#)
- [If](#)
 - [If composite](#)
- [For in](#)
- [Scan by](#)
 - [Parcourir un tableau à deux dimensions](#)
- [While](#)
 - [While let](#)
- [Do until](#)
- [Do case](#)
- [Syntaxe étendue de do](#)
- [Break et continue](#)
- [Enum](#)
- [Indexation](#)
- [Intervalle](#)
- [Intervalles dans une liste](#)
- [Text](#)
 - [Méthodes de text](#)
- [Variable dynamique](#)
- [Séquence et liste](#)
- [Tableau](#)
 - [Créer un tableau](#)
 - [Indexer un tableau](#)
 - [Itérateur](#)
 - [Utiliser un tableau comme pile](#)
 - [Intervalle dans un tableau](#)
 - [Opérateurs de tableau](#)
 - [Tableau à plusieurs dimensions](#)
 - [Contenu d'un tableau en PHP, step by step](#)
- [Dictionnaire](#)
 - [Créer un dictionnaire](#)
 - [Indexer un dict](#)
 - [Intervalle dans un dict](#)
- [Méthodes de array et dict](#)
- [Tableaux typés](#)
 - [Constructeur et littéral](#)
 - [Tableau d'entiers](#)
 - [Tableau de textes](#)
 - [Tableau de real, natural, number](#)
 - [Tableau d'objets](#)

- [Assignement et conversion](#)
- [Tableau typé et dyn](#)
- [Assignement direct](#)
- [Limitations et compatibilité](#)
- [File](#)
 - [Read](#)
 - [Write](#)
- [Dir](#)
- [La structure de contrôle error](#)
- [Scopes](#)
- [Extern, externe variables, constantes et fonctions](#)
 - [Variables et constantes](#)
 - [Fonctions externes](#)
 - [Classe externe](#)
 - [Types externes](#)
 - [Insérer du code cible](#)
- [Class](#)
 - [Définir une classe](#)
 - [Constructeur et instance](#)
 - [Méthodes et attributs statiques](#)
 - [Héritage](#)
 - [Surcharge](#)
- [XML statique](#)
 - [XML light](#)
 - [Définir un document XML](#)
 - [Chargement et sauvegarde](#)
 - [Assignement XML](#)
 - [Itérateur XML](#)
 - [Utiliser les données](#)
 - [Utiliser les attributs](#)
 - [XML et fonction](#)
- [XML dynamique](#)
- [Méthodes XML](#)
- [Include](#)
- [Define](#)
 - [Utiliser une fonction comme variable](#)
 - [Utiliser un type externe](#)
- [Utiliser les librairies standards](#)
 - [Librairies PHP](#)
 - [Librairies C](#)
- [Fonctions usuelles](#)
- [XML intégré \(Scriptol C++ only\)](#)
- [Appendice I: Utiliser l'API Java API avec Scriptol](#)
- [Appendice II: Gestion des exceptions](#)
- [Appendice III: Langues étrangères](#)
- [Appendice IV: Syntaxe obsolète](#)
- [Index](#)

Généralités

Le nouveau langage de programmation Scriptol peut être utilisé avec un interpréteur et des compilateurs.

Une version Windows et Unix existe pour chaque compilateur.

L'objectif du langage est d'être simple, naturel et donc de réduire le risque d'erreurs. Scriptol signifie: "Scriptwriter Oriented Language" ou "langage de scénariste" en français. C'est un langage universel conçu pour produire des pages web dynamiques, des scripts et des applications avec interface graphique. Ce peut être aussi un langage de script pour les applications traitant ou produisant des documents XML. Scriptol peut être imbriqué dans une page HTML et est converti en la même page avec du code PHP, prêt pour le Net et totalement portable. Le langage a été défini selon sept règles et une page explique en détail pourquoi utiliser Scriptol.

Node: [Au sujet de ce manuel](#), Next: [Interpréter un programme scriptol](#), Previous: [Généralités](#), Up: [Top](#)

Au sujet de ce manuel

Veillez noter que les symboles [] inclus dans la syntaxe des instructions ne font pas partie de la syntaxe et indiquent un élément optionel, sauf pour les indices et intervalles qui utilisent précisément ces caractères.

Node: [Interpréter un programme scriptol](#), Next: [Compiler un programme Scriptol en PHP](#), Previous: [Au sujet de ce manuel](#), Up: [Top](#)

Interpréter un programme scriptol

La commande est:

si [options] fichiersource

Si des erreurs sont trouvées, elle sont affichées et le programme ne s'exécute pas.

Options de l'interpréteur:

```
aucune:  interprète et exécute un fichier source Scriptol.  
-v:     verbeux, affiche des informations détaillés.  
-q:     silencieux (quiet), aucun message.  
-f:     force, ignore les erreurs et exécute le programme.
```

Node: [Compiler un programme Scriptol en PHP](#), Next: [Compiler un programme Scriptol en binaire](#), Previous: [Interpréter un programme scriptol](#), Up: [Top](#)

Compiler un programme Scriptol en PHP

Sous Windows, la commande est:

solp [options] nomfichier

Si le source est déjà compilé, le programme est interprété, sinon il est compilé avant d'être interprété s'il n'y a pas d'erreur.

Tant que le source a des erreurs une nouvelle commande "solp nomfichier" va le compiler. Si le source est une page HTML, il n'est pas exécuté après compilation. Sous Linux, la commande pour compiler est: solp options nomfichier

suivie de:

php -q nomfichier.php pour l'exécuter.

Options du compilateur Scriptol-php:

```
aucune: compile un script en PHP ou le lance si déjà compilé.
-w      compile du code inclus dans une page HTML et crée un fichier PHP.
-t      teste seulement, n'exécute pas.
-r      force l'exécution. Compile si besoin seulement.
-b      recompile le fichier et fichiers inclus. Le programme n'est pas lancé.
-t      invoque la traduction du source s'il est en français.
-v      affiche plus d'informations lors de la compilation.
-q      pas de message de compilation.
-4:     compile selon la syntaxe PHP 4 (les classes propres à PHP 5 ne sont pas utilisables).
```

Ces lettres peuvent être combinées en une seule option. Ainsi -be est équivalent à -b -e

Node:Compiler un programme Scriptol en binaire, Next:[Mon premier programme](#), Previous:[Compiler un programme Scriptol en PHP](#), Up:[Top](#)

Compiler un programme Scriptol en binaire

La commande est:

solc [options] fichiermain

Le fichier main est celui qui contient la fonction main(). Les dépendances sont calculées par le compilateur qui sait ce qui doit être compilé ou non.

Options du compilateur Scriptol-C++:

```
-b      compiler tous les fichiers objets.
-c      compiler tous les sources en C++ seulement.
-e      lier les objets en un exécutable.
-r      lancer l'exécutable.
-t      invoquer la traduction.
-v      informations de compilation détaillées.
-q      pas de messages.
```

Node:Mon premier programme, Next:[Projet Scriptol](#), Previous:[Compiler un programme Scriptol en binaire](#), Up:[Top](#)

Mon premier programme

Un source Scriptol source est juste un fichier de texte contenant des commandes inside, une par ligne.

Exemples:

```
print x * 2
print x + (y / 2)
```

Avec l'éditeur fourni ou un éditeur quelconque, taper cette ligne:

```
print "Bonjour le monde"
```

Sauver dans le fichier `bonjour.sol`.

Cliquez sur la commande "interpreter" dans le menu "Tools". La phrase "Bonjour le monde", doit s'afficher.

Si vous travaillez dans une fenêtre de ligne de commande (MS-DOS sous Windows, console sous Unix), tapez simplement:

```
si bonjour          ...ou
solp bonjour
```

Vous pouvez aussi construire un exécutable avec la commande:

```
solc -bre bonjour
```

et lancez le programme `bonjour.exe` généré.

Il est possible de travailler à la fois avec l'éditeur et une fenêtre de console.

Node:Projet Scriptol, Next:[Fichier source Scriptol](#), Previous:[Mon premier programme](#), Up:[Top](#)

Projet Scriptol

Aucun fichier projet n'est nécessaire.

En Java ou PHP, un programme est un source qui inclut d'autres sources. Scriptol conserve cette structure.

Un source scriptol doit avoir une instruction "include" pour chaque fichier dont le contenu est utilisé. Le compilateur calcule les dépendances sur plusieurs niveaux d'inclusions, et parmi les inclusions mutuelles.

Node:Fichier source Scriptol, Next:[Caractéristiques du langage](#), Previous:[Projet Scriptol](#), Up:[Top](#)

Fichier source Scriptol

Un fichier Scriptol doit avoir l'extension "sol", et sera converti en un fichier d'extension php, ou cpp et hpp, ou un exécutable.

Cela la forme du source, on aura un script ou une application.

Script

Le source d'un script est une séquence d'instructions et de définitions de fonctions ou de classes. Il peut

inclure d'autres fichiers contenant des déclarations de fonctions, de variables, de classes.
Un script ne peut contenir de fonction "main" (pour le compilateur, il est lui-même une fonction "main").

Application

Un fichier d'application contient seulement des déclarations de fonctions, variables et classes. Le source principal est celui que l'on passe en ligne de commande du compilateur, il doit contenir une fonction "main", et une commande d'appel de cette fonction, comme ci-dessous:

```
int main()  
    ... instructions ...  
return 0  
  
main()    // lancement du programme.
```

Node:Caractéristiques du langage, Next:[Page HTML](#), Previous:[Fichier source Scriptol](#), Up:[Top](#)

Caractéristiques du langage

Scriptol peut être défini comme:

- orienté objet.
- orienté XML (un document XML peut être une structure de donnée du langage).
- universel: utilisable pour faire des scripts, des applications ou pages web dynamiques.
- naturel: les types viennent de la culture et non du matériel (number, text, real, ...).
- style XML pour la syntaxe.
- ayant des structures nouvelles et plus puissantes.
- traitement de liste avec tableaux et dictionnaires.
- compatible avec C, PHP, Java (peut utiliser leurs bibliothèques).

C'est un langage clair grâce à:

- une syntaxe simple.
- les instructions terminées par la fin de ligne.
- un opérateur unique pour les intervalles, les sous-listes.
- une syntaxe similaire pour toutes les structures.

Sensibilité à la casse:

- On ne peut utiliser un même mot en majuscules et minuscule.
- les mots-clés sont en minuscule.
- les identificateurs sont sensibles à la casse, mais on ne peut redéfinir un identificateur avec une casse différente.

Identificateurs:

- taille jusqu'à 255 caractères ou moins selon le langage cible.
- commence par une lettre majuscule ou minuscule.
- suivi de lettres, caractères de soulignements, ou chiffres.

Nombres:

- les entiers sont signés sur 32 bits (comme "int" en C).
- les naturels sont non-signés sur 64 bits.
- réels, nombres ("number") sont en virgule flottante sur 64 bits ("double" en C).

Conversion (cast):

- utilisation de méthodes.

Garbage-collector (allocation de mémoire):

- aucun besoin d'allouer ou libérer la mémoire.

Orienté-objet:

- les primitives sont des objets et ont des méthodes.

- les littéraux sont des objets et ont des méthodes.

- héritage simple.

- surcharge de méthodes (version C++ seulement pour l'instant).

- constructeurs. Pas de destructeurs.

Orienté-XML:

- des documents XML peuvent être inclus dans un source Scriptol, XML est une structure de donnée du langage .

- instances de documents XML.

Node:Page HTML, Next:[Instruction](#), Previous:[Caractéristiques du langage](#), Up:[Top](#)

Page HTML

Pour que le code Scriptol puisse être inséré dans du HTML, il doit être inclus dans les balises suivantes:

```
<?sol      ...code...      ?>
ou <?script language=scriptol>  ...code...  </script>
```

'scriptol' ou "scriptol" sont reconnus également. Le mot-clé "script" peut être requis pas certains éditeurs HTML. La dernière ligne d'un script Scriptol doit être terminée par un point-virgule ou un saut de ligne, avant le symbole ?>

Tester les pages HTML Si vous avez installé un serveur comme Apache ou Xitami ou Windows Server sur votre ordinateur, et l'avez configuré pour reconnaître l'extension php, votre code sera traité comme sur le web, une fois compilé en PHP. Autrement, il vous rediriger dans un fichier test.HTML la page dynamique, et ensuite exécuter le code PHP:

```
solp -w mapage
php mapage.php > test.HTML
```

Node:Instruction, Next:[Commentaire](#), Previous:[Page HTML](#), Up:[Top](#)

Instruction

Une instruction est terminée par un point virgule ou par la fin de la ligne. Quand une instruction dépasse la largeur d'une ligne, elle est concaténée avec la suivante pourvu qu'elle soit terminée par une virgule, un opérateur ou tout autre symbole qui ne peut terminer une instruction. Aucun symbole n'est requis pour continuer une instruction sur la ligne suivante. Plusieurs instructions sur une même ligne sont

séparés par un point virgule. Cela peut être utile si vous placez du code Scriptol dans une page HTML et si l'éditeur concatène les lignes!

Node:Commentaire, Next:[Symboles](#), Previous:[Instruction](#), Up:[Top](#)

Commentaire

Commentaires d'une seule ligne: ` le commentaire va du symbole jusqu'à la fin de la ligne. // le symbole du C++ est reconnu aussi. Les commentaires ne sont pas conservés dans le langage cible. " le symbole doublé rend un commentaire persistant dans le code cible. Un commentaire de plusieurs lignes commence par /* et se termine par */

Node:Symboles, Next:[Identifieurs et mot-clés](#), Previous:[Commentaire](#), Up:[Top](#)

Symboles

Le langage Scriptol n'utilise jamais le même symbole pour des usages conceptuellement différents.

+	addition
-	sousstraction
*	multiplication
/	division
^	puissance
=	opérateur d'assignement.
<	moins que.
>	supérieur à.
<=	inférieur ou égal.
>=	supérieur ou égal.
:=	assignement conditionnel.
;	est un terminateur d'instruction.
,	sépare les éléments d'un initialiseur ou d'un tuple.
:	attache un corps à une en-tête, une valeur à une clé...
.	associe la référence d'une méthode à un objet.
()	regroupe des sous-expressions, les arguments d'une fonction.
..	sépare les limites d'un intervalle.
--	intervalle avec la limite supérieure non incluse.
[]	contiennent un index ou un intervalle.
?	sépare condition et action dans une structure d'une seule ligne.
<>	opérateur de différence.
&	et binaire ou intersection de tableaux.
	ou binaire ou union de tableaux.
<<	décalage à gauche.
>>	décalage à droite.
and	et logique (sur valeurs booléennes).
or	ou logique.
not	négation logique.
mod	modulo (C utilise %).
~~	un code à insérer directement dans le code généré.

Node:Identifieurs et mot-clés, Next:[Variables ou primitives](#), Previous:[Symboles](#), Up:[Top](#)

Identifieurs et mot-clés

Les identifieurs sont les noms de variables, fonctions et objets. Ils commencent par une lettre, suivie de lettres, chiffres ou caractères de soulignement. On ne peut les différencier par la casse: on ne peut donner le même nom à deux objets différents, l'un en majuscules, l'autre en minuscules. Les mot-clés sont les mots réservés du langage, ils sont en minuscules.

Node:Variables ou primitives, Next:[Litéraux](#), Previous:[Identifieurs et mot-clés](#), Up:[Top](#)

Variables ou primitives

Les primitives sont des objets de base. Quand elle sont les arguments d'une fonction, la fonction utilise une copie de leur contenu, de leur valeur, et ne modifie par l'original, tandis qu'elle utilise un alias pour les autres objets.

Les primitives de Scriptol sont celles-ci:

number	toute sorte de nombre (jusqu'au double du C++).
int	un nombre arrondi à la partie entière. 32 bits. -1 vaut nil.
integer	identique.
natural	entier non signed de 64 bits. 0 vaut nil.
real	un nombre avec des décimales. -1 vaut nil.
boolean	la valeur vrai ou faux.
text	une chaîne de caractères. "" vaut nil.
array	une liste dynamique indexée d'objets.
dict	une liste associative formée de couples clé:valeur.

D'autres types speciaux existent:

dyn	élément générique de array ou valeur de dict.
file	un fichier.
dir	un répertoire.
byte	type externe de C.
cstring	utilisé dans les déclarations externes en langage C (traduit par char *).
*	désigne un pointeur de C associé à un type.

Les primitives ont des constructeurs comme les autres objets avec différents argument, ce qui permet de les utiliser pour des conversions de type.

Syntaxe:

```
int(int | number | text)
```

Exemples:

int("123")	...représente le nombre 123
text(10)	...représente la chaîne "10".

Les primitives sont des variable sauf si elles sont explicitement déclarées comme constantes, avec le mot-clé "constant".

Node:Litéraux, Next:[Guillemets et échappement](#), Previous:[Variables ou primitives](#), Up:[Top](#)

Littéraux

Les nombres littéraux s'écrivent:

123	entier (int)
123n	naturel (natural)
123.0	réel (real)
0.123	réel (real)
1.2e10	réel (real)
0xf8	hexadécimal
true/false	booléen (boolean)

Node:Guillemets et échappement, Next:[Variable dans une chaîne](#), Previous:[Littéraux](#), Up:[Top](#)

Guillemets et échappement

Un texte littéral s'écrit: "untexte" ou 'untexte'.

Exemple:

```
print "abc$xyz"  
print 'abc$xyz'
```

dans les deux cas affiche abc\$xyz.

Codes d'échappement

Pour les caractères qui ne peuvent être placés directement dans un texte, on utilise le code spécial: \ Par exemple pour afficher un guillemet, on écrira: "abc\"def"

\"	insère un guillemet double.
\'	insère un guillemet simple.
\n	insère a retour à la ligne.
\t	insère une tabulation.
\\	insère l'anti-slash lui-même.

Supposons que vous voulez assigner la variable a avec exactement ce texte: ma "jolie" voiture. Vous ne pouvez le mettre entre guillemets: a = "ma "jolie" voiture" Le compilateur ne saura comment le traiter! Un code special est utilisé pour dire que les guillemet sont dans le texte plutôt qu'ils ne le délimitent, c'est \ a = "ma \"jolie\" voiture" Vous pouvez aussi alterner les types de guillemets: a = "ma 'jolie' voiture" ou a = 'ma "jolie" voiture'

Multi-lignes

Vous pouvez aussi avoir un texte répartis sur plusieurs lignes enclos dans les symboles "~~".

Exemple:

```
x = ~~  
  ligne 1  
  ligne 2  
  ~~
```

Le retour à la ligne peut être direct ou introduit par un code d'échappement.

```
x = "un\ndeux"           ...valide
x = "un
deux"                   ... non valide

x = ~~un
deux~~                  ... valide
```

Node:Variable dans une chaîne, Next:[Déclaration](#), Previous:[Guillemets et échappement](#), Up:[Top](#)

Variable dans une chaîne

Si une chaîne est entre double guillemets, les séquences de caractères commençant pas \$ sont des noms de variables.

```
text xyz = "def"
text t = "abc$xyz"
```

De même, les symboles {} ont une signification particulière pour l'interpréteur PHP, mais cela ne fait pas partie du langage Scriptol.

Les textes entre guillemets simples, et précédés de \$ ne sont pas interprétés du tout.

```
text t = 'av$er\n'
```

affichera: av\$er\n

Node:Déclaration, Next:[Constante](#), Previous:[Variable dans une chaîne](#), Up:[Top](#)

Déclaration

La déclaration d'une variable a la forme:

```
type nom [ = valeur ]

int x
text t = "demo"
```

Pourvu qu'elles aient toutes le même type, il est possible de déclarer et initialiser plusieurs variables à la fois.

La syntaxe est alors:

```
type nom [= value], name [= value], etc...

int x = 1, y, z = 2
```

On ne peut mêler plusieurs types dans une seule déclaration. On ne peut faire d'assignements multiples, comme il est décrit plus loin, dans une déclaration.

La forme suivante n'est pas valide:

```
int x, y = 0, 0
```

Constante

Le modifieur "constant" dénote une primitive dont la valeur ne peut changer. Une valeur doit être assignée lors de la déclaration.

Syntaxe et exemples de déclarations de constantes:

```
constant type nom = valeur

constant int X = 1
constant boolean B = true
constant text T = "texte quelconque"
```

Les constantes sont usuellement en majuscules.

Il y a des constantes prédéfinies qui sont des mot-clés du langage:

true	correspond à une expression vraie.
false	le contraire.
zero	la valeur 0.
nil	objet non trouvé dans une séquence (Not In List).
null	assigné à un objet quand il n'est pas encore initialisé.

Nil et null

Le mot-clé nil réfère au contenu d'un objet, tandis que null remplace une adresse. null signifie "sans valeur", ou "déclaré mais non défini", tandis que nil signifie "non trouvé" ou "vide". Le mot-clé null est converti en "null" en PHP et "NULL" en C++;

Si null est assigné à une identifieur, il ne peut être référencé ou donc utilisé jusqu'à ce qu'on lui assigne une valeur. On peut seulement le comparer dans une condition mot-clé null.

Nil n'est pas une vraie valeur mais plutôt une construction du langage. Voici les valeurs de nil en différents contextes...

Quand du code C++ est généré, nil est remplacé par ces valeurs:

boolean	false
int	-1
natural	0
real	-1
text	""
array	{}
dict	{}
file	NULL
dir	NULL
objet	NULL
dyn	selon le contenu.

Quand on assigne nil à un intervalle dans un tableau, nil supprime cet intervalle du tableau.

Quand du code PHP est généré, nil est remplacé par ces valeurs:

```
retour d'une fonction:   false
dans une expression:    false
assignement:            comme C++
```

Node:Assignement, Next:[Assignement simple](#), Previous:[Nil et null](#), Up:[Top](#)

Assignement

Node:Assignement simple, Next:[Assignement augmenté](#), Previous:[Assignement](#), Up:[Top](#)

Assignement simple

La syntaxe d'un assignement simple est:

```
identifieur = expression
```

La valeur assignée doit avoir le type de la variable:

```
int i = 10
int j = "343"           ... incorrect
```

Cependant, les constructeurs de primitives permettent de faire des conversions.

Node:Assignement augmenté, Next:[Assignements multiples](#), Previous:[Assignement simple](#), Up:[Top](#)

Assignement augmenté

Lorsque une opération est effectuée sur une variable, et le résultat assigné à la même variable, il est possible de simplifier l'instruction, en assignement augmenté (d'une opération) .

La syntaxe d'un assignement augmenté est:

```
identifieur opérateur expression
```

Par exemple, ajouter 10 à x et mettre le résultat dans x. Au lieu d'écrire `x = x + 10`, on écrira plus simplement: `x + 10`

Les opérateurs possibles sont: + - * / mod << >> & | ^ #

Exemples:

```
a = 1           ... donne à a la valeur 1.
a + 1          ... ajoute 1 à a.
x * y         ... remplace le contenu de x par le résultat x * y.
```

```
a * (x + 2)      ... multiplie a par l'expression.  
a = a * (x + 2) ... comme ci-dessus.  
a & b          ... remplace le tableau a par l'intersection des tableaux a et  
b.
```

Noter que dans un test de condition, `x + 10` retourne simplement le résultat de l'addition de 10 au contenu de `x`, sans modifier `x`. Dans une instruction augmentée, ce résultat est affecté à `x`.

```
if x + 1 : ...      ici x n'est pas modifié.
```

Node:Assignements multiples, Next:[Assignement conditionnel](#), Previous:[Assignement augmenté](#), Up:[Top](#)

Assignements multiples

Un tuple de valeurs peut être assigné à un groupe de variable. Syntaxe et exemple:

```
nom [, nom]* = expression [, expression]*  
  
x, y, z = 1, x2, 8
```

Le nombre d'expressions à droite doit correspondre au nombre d'expressions à gauche. Sauf dans le cas où une seule expression figure à droite, elle sera assignée à chaque variable du groupe. Noter qu'une fonction peut retourner plusieurs valeurs et sera assignée à un groupe de variables. Exemples:

```
x, y, z = 0  
a, b = mafonction()
```

Au contraire de la déclaration multiple, un assignement multiple peut se faire avec des variables de types différents.

Exemple:

```
int x  
text t  
x, t = 1000, "abc"
```

L'assignement multiple peut s'utiliser avec les éléments d'un tableau. Les tableaux ont une taille dynamique et donc, le nombre de cible n'a pas besoin de correspondre au nombre d'éléments, et si le tableau n'a qu'un seul élément, seule la première variable sera assignée.

Exemple:

```
x, y, z = array(1, 2, 3, 4)
```

Est équivalent à: `x = 1, y = 2, z = 3`.

```
x, y, z = array(1)
```

Est équivalent à: `x = 1`.

Vous pouvez assigner plusieurs variables à la fois avec: - une valeur ou expression.

- un array
- les valeurs d'un dict (utiliser la méthode `values()`).
- un appel de fonction.

- un tuple de valeurs ou expressions séparées par des virgules.

Dans le cas d'array ou dict, de tuple, de fonction retournant plusieurs valeurs, on affecte aux variables 1 à n les éléments 1 à n, dans le même ordre. Si le nombre ne correspond pas, c'est une erreur.

Si la fonction ne retourne qu'une seule valeur, la même valeur est assignée à toutes les variables à gauche.

Node:Assignement conditionnel, Next:[Opérateurs](#), Previous:[Assignements multiples](#), Up:[Top](#)

Assignement conditionnel

Cet assignement spécial a pour but d'assigner une propriété ou autre variable qui a déjà une valeur par défaut, quand une nouvelle valeur est donnée. Le symbole := assigne conditionnellement une variable, si la valeur à assigner ne vaut pas nil.

Le drapeau error est mis à "vrai" si l'expression à affecter vaut nil.

Exemple:

```
x := z
error ? print "z est nil"
```

L'instruction ci-dessus équivaut à:

```
if z <> nil
  x = z
else
  print "z est nil"
/if
```

Node:Opérateurs, Next:[Expression](#), Previous:[Assignement conditionnel](#), Up:[Top](#)

Opérateurs

Les opérateurs de comparaison sont:

```
=      (égal)
<      (inférieur)
>      (supérieur)
<=     (inférieur ou égal)
>=     (supérieur ou égal)
<>     (différent)
```

L'opérateur "in" teste la présence d'un élément dans une séquence: texte dans un texte, objet dans un tableau, valeur dans un intervalle.

```
if "a" in line print "dans le texte"
if x in 1..10 print "dans l'intervalle"
```

Les opérateurs binaires sont:

```
&      (et)
|      (ou)
^      (ou exclusif)
~      (non)
```

<< (décalage à gauche)
>> (décalage à droite).

Les opérateurs de tableaux sont:

& (intersection)
| (union)
^ (complément d'intersection).

Précédence

Les opérateurs unaires ont la priorité sur les opérateurs binaires. Entre opérateurs de même arité, la priorité est donnée par les parenthèses.

Node:Expression, Next:[Précédence](#), Previous:[Opérateurs](#), Up:[Top](#)

Expression

Une expression est une combinaison de valeurs et d'opérateurs.

Expressions arithmétiques

Ce sont des valeurs arithmétiques ou des appels de fonctions combinés avec ces opérateurs:

+
-
*
/
^ puissance
mod modulo, retourne le reste d'une division...

Exemple:

```
print 100 mod 3    ... affiche 1 le rester de 100 divisé par 3.
```

Expressions relationnelles

Les expressions arithmétiques peuvent être comparées grâce à un ensemble d'opérateurs:

= égal
< inférieur à
> supérieur à
<= inférieur ou égal
>= supérieur ou égal
<> différent

La comparaison retourne une valeur booléenne: true ou false (vrai ou faux). Cette valeur booléenne peut être assignée à une variable booléenne ou utilisée comme condition d'une structure de contrôle, telle que "if" notamment.

Expressions logiques

Une expression logique est une combinaison de valeurs booléennes ou d'expressions (relationnelles ou logiques) et d'opérateurs logiques.

Les opérateurs logiques sont:

```
and      et
or       ou
not      non
```

L'expression "and" retourne vrai si les deux termes sont vrais, faux sinon.

L'expression "or" retourne vrai si un des deux termes est vrai, faux si les deux sont faux.

Exemple:

```
boolean x = true
boolean y = false
if x and y print "vrai"; else print "faux"           ... doit afficher faux
not(x or y)           ... est faux
x and not y           ... est vrai
```

Expressions binaires

Les opérateurs binaires sont ceux que la plupart des langages utilisent:

```
&  et binaire
|  ou binaire
^  ou exclusif
~  négation binaire
<< rotation à gauche qui équivaut à multiplier par 2
>> rotation à droite qui équivaut à diviser par 2
```

Expressions textuelles

Les opérateurs de texte sont:

```
=, <, >, <=, >=  comparaison de deux textes.
+                concaténation de deux textes.
[]              indexation, slicing ou splicing (voir chapitre Text).
in              teste si un text fait partie d'un autre.
```

Exemple:

```
text t = "préfixe"
print t + "suffixe"
...affichera: préfixesuffixe
```

Expressions de listes dynamiques

Les opérateurs de listes (array, dict) sont:

```
= < > <= >= <>  compare les valeurs de deux listes.
+                concatène deux listes (des doubles peuvent en résulter).
-                enlève d'une liste les éléments d'une autre (sauf doubles).
[]              indexation, slicing or splicing (voir Array et Dict).
in              teste si un élément est dans la liste.
&                intersection.
|                union.
^                complément d'intersection.
```

L'intersection de deux listes retourne les éléments communs aux deux. L'union de deux listes retourne l'ensemble de leurs éléments. Les éléments appartenant aux deux ne sont gardés qu'une fois. Si une des listes a déjà un élément en double, seule la première occurrence est conservée.

Node:Précédence, Next:[Fonction](#), Previous:[Expression](#), Up:[Top](#)

Précédence

Quelques langages de programmation ont institué des règles de précédence, ainsi, quand les parenthèses sont omises, on sait quel terme est concerné par chaque opérateur. Bien que la précédence ait été construite dans le parseur Scriptol, des erreurs de messages seront envoyés si les parenthèses sont omises, car elles sont nécessaires à la lisibilité.

Le seul cas où la précédence est admise sans parenthèse est pour les opérateurs unaires: not, ~ Un opérateur unaire s'applique toujours au terme qui le suit, donc pour l'appliquer à une expression il faut la mettre entre parenthèses. Exemple:

```
if not a and b
if not (a and b)
```

L'opérateur not est associé à l'objet "a" dans le premier, dans le second on a la négation de l'expression.

Dans les assignements composé (voir plus loin) le premier opérateur concerne la variable à assigner avec l'expression à sa droite.

Node:Fonction, Next:[Alias](#), Previous:[Précédence](#), Up:[Top](#)

Fonction

Une fonction est définie avec une interface qui indique comment l'appeler, un ensemble d'instructions, et se termine avec le mot-clé "return".

Le type de retour est requis et le type des arguments aussi. On utilise "void" si la fonction ne retourne rien.

Syntaxe:

```
type [,type]* nom ( argument [, argument]* ) [:]
... instructions ...
return [expression [, expression]*]
```

Exemple:

```
int multiply(int x, int y)
    int z
    z = x * y
return z
```

Cela peut être écrit plus simplement:

```
int multiply(int x, int y)
return x * y
```

Le corps de la fonction est une liste d'instructions, incluant des instructions "return" si besoin. La fin de définition est un return avec zéro, une, ou plusieurs valeurs.

```
return
```

```
return x
return x, y, 5
```

Si la fonction retourne plusieurs valeurs, elle doit avoir plusieurs types de retour et l'instruction de retour à plusieurs paramètres (dans le même ordre que les types de retour).

Exemple:

```
text, int, int coordonnees(int num)
    int x = matable[num].x
    int y = matable[num].y
return "coordonnées=", x, y
```

Appel de fonction

L'appel d'une fonction peut assigner zéro, une, ou plusieurs variables.

```
mafunc()
a = mafunc()
a,b,c = mafunc()
```

Node:[Alias](#), Next:[Valeurs par défaut](#), Previous:[Fonction](#), Up:[Top](#)

Alias: passage par valeur ou par référence

Quand une fonction a des objets comme arguments, les noms des arguments sont des alias des objets, et toute modification dans la fonction sont effectués en fait sur les objets originaux. Par défaut, les primitives sont des copies et les objets des alias. On peut le changer avec un modifieur: "alias": le nom de la primitive devient un alias de l'original.

```
void mafunc(number x)
    x + 1
    print x
return

void foncalias(alias number x)
    x + 1
    print x
return

number y = 5
mafunc(y)
print y
... doit afficher 6 puis 5

foncalias(y)
print y
... doit afficher 6 puis 6.
```

Node:[Valeurs par défaut](#), Next:[Scope et fonction](#), Previous:[Alias](#), Up:[Top](#)

Valeurs par défaut

Assigner une valeur par défaut permet d'omettre un paramètre à l'appel d'une fonction. La syntaxe

complète de l'interface est:

```
type [,type]* nom(type nom [= expression] [, type nom [= expression]]* )
```

Exemple:

```
int increment(int x, int y = 1)
    x + y
return x
print increment(10, 5)          ...affichera: 15
print increment(10)            ...affichera: 11
```

La valeur par défaut 1 a été utilisée pour remplacer le paramètre manquant.

Si l'interface d'une fonction a plusieurs arguments avec une valeur par défaut, vous ne pouvez en omettre un lors de l'appel sans omettre tous les suivants. Les arguments ne peuvent être reconnus par leur type.

Si ce que vous voulez, c'est écrire une fonction ayant un nombre et des types de paramètres variable, vous devez plutôt utiliser un tableau ou dictionnaire.

Exemple:

```
void param(dict parlist)
    taille = parlist["taille"]
    nom = parlist["nom"]
    valeur = parlist["valeur"]
return
```

Dans cet exemple, les variables "taille", "nom", "valeur" sont globales, et elle sont assignée par le contenu du dict en argument.

Pour que cet exemple soit parfait en fait, on devrait utiliser des assignements conditionnels, grâce à un test. Ou grâce au symbole d'assignement conditionnel :=

```
x := y
```

x est assigné par la valeur de y seulement si y n'est pas nil.

Node:Scope et fonction, Next:[Mode silencieux](#), Previous:[Valeurs par défaut](#), Up:[Top](#)

Scope et fonction

Le scope est un niveau de l'espace de visibilité des objets déclarés. Une fonction ouvre un nouveau scope pour toutes les variables déclarées à l'intérieur. Si la fonction est dans le scope global, toutes les variables globales compilées avant la fonction sont visibles dans la fonction.

On ne peut redéclarer dans la fonction une variable avec le même nom qu'une variable globale.

Si la fonction est une méthode d'une classe, toutes les variables déclarées dans la classe avant la fonction sont visibles dans la fonction.

Les objets déclarés dans la fonction sont visibles dans les structures de contrôle à l'intérieur de la fonction.

Node:Mode silencieux, Next:[Main](#), Previous:[Scope et fonction](#), Up:[Top](#)

Le mode silencieux

Il est possible d'empêcher que l'interpréteur PHP n'affiche les messages d'erreur ou d'avertissement en faisant précéder l'appel d'une fonction du symbole @.

Le symbole est ignoré par le compilateur binaire.

```
@mafonction(x, y)
```

Node:Main, Next:[Print et echo](#), Previous:[Mode silencieux](#), Up:[Top](#)

La fonction "main"

Il est utile de pouvoir passer des paramètres à un programme à partir de la ligne de commande.

Un script Scriptol (comme PHP) utilise la variable PHP \$argv". Pour imiter la façon dont C++ passe des paramètres à un programme, créer une fonction "main" et appelez-la avec \$argv en argument (et \$argc si besoin):

```
int main(int argnum, array arglist)
    print argnum, "arguments"
    scan arglist
        print arglist[]
    /scan
return 0

main($argv, $argc) // argv and argc sont des variables PHP externes.
```

La fonction main, soit n'a aucun argument, soit en a deux:

```
int main()
int main(int, array)
```

Node:Print et echo, Next:[Input](#), Previous:[Main](#), Up:[Top](#)

Print et echo

Echo

Affiche une expression, ou une liste d'expressions séparées par des virgules, sans espaces entre elles et sans retour à la ligne à la fin (comme le fait print).

Syntaxe: echo expression [, expression]

Exemple:

```
x = 5
y = 20
z = 1
echo "values", x, y / 2
echo z
```

Ce sera affiché à l'écran comme ceci:

values5101

Exemple:

```
echo "demo", 5  
echo "next"
```

affiche: demo5next

Print

Un affichage plus élaboré est obtenu avec l'instruction print. Les expressions sont séparées et on va à la ligne en fin d'instruction.

Syntaxe: print expression [, expression]

Exemple:

```
print "demo", 5
```

affiche: demo 5

Une instruction print simple, sans expression, envoie un retour à la ligne.

Exemple:

```
print
```

Node:Input, Next:[Structures de contrôle](#), Previous:[Print et echo](#), Up:[Top](#)

Input

Pour entrer du texte à partir du clavier, utiliser la commande input.

Un texte peut être affiché avant la saisie du texte.

Exemple:

```
text name  
input name  
print name
```

Exemple:

```
input "who are you? ", name
```

La variable à assigner doit être déclarée avant la commande input, ce peut être un text ou un nombre de type quelconque.

Node:Structures de contrôle, Next:[If](#), Previous:[Input](#), Up:[Top](#)

Structures de contrôle

Les structures de contrôle sont:

```
if
  if else /if.
  if une instruction.
  if composite.
for
  for /for.
  for une instruction.
  - options: in intervalle, in séquence.
scan
  by fonction.
  scan une instruction.
  scan /scan.
while
  while let.
  while une instruction.
  while forever
do while
  do until.
  do /do condition.
  do case / else / always /do.
enum
  enum simple.
  enum dictionnaire.
error
  error /error.
  error une instruction.
```

Scriptol a une syntaxe différente pour les structures de contrôle d'une instruction, ou multi-lignes.

Une structure de contrôle d'une seule instruction a la forme:

```
mot-clé-de-structure expression let instruction
ou mot-clé-de-structure expression ? instruction
ou mot-clé-de-structure expression instruction commençant par un mot-clé.
```

Il n'y a pas d'autre marqueur de fin que la fin de ligne. L'instruction est basique et ne peut être un autre construct.

L'instruction "let" est toujours la dernière partie d'une structure de contrôle et peut être la seule. Le symbole "?" est simplement une abbréviation de "let". Let est requis si l'instruction est un assignement ou un appel de fonction, mais optionel si un mot-clé le suit. Si "else" ou "let" complète la structure de contrôle sur la même ligne, l'instruction doit être terminée par un point-virgule.

Une structure de contrôle multi-lignes a la forme:

```
nom-structure expression [:]
... instructions ...
/nom-structure
```

Le deux-points est optionnel (sauf si une instruction est mise sur la même ligne).

Exemple avec séparateurs:

```
if a = b : print t ; /if
```

If

Une structure de contrôle conditionnelle d'une ligne à la forme:

```
if expression-booléene let instruction
[else instruction]
ou:
if condition let instruction ; else instruction
```

Cela se lit ainsi: si condition vraie? alors action; sinon action

Scriptol utilise soit le mot-clé "let" ou "?" quand l'action est une instruction unique ou ":" ou un retour à la ligne quand c'est une bloc d'instructions.

Exemples:

```
if a = 5 ? break
if a < 5 ? print "moins que"
    else print "plus que ou égal"
if a = 1 ? print "un"; else print "plusieurs"
```

Multi-lignes

Syntaxe:

```
if condition-booléenne [:]
    ... instruction ...
else
    ...optionnel
    ... instructions ...
/if
```

N.B.: Le symbole deux-points est optionnel après la condition, comme le point-virgule après une instruction, mais il est requis quand on concatène les lignes.

Exemple:

```
if (x + y) > 8
    print "> 8"
    print x
else
    print "<= 8"
/if

if (x + y) > 8 : print "> 8" ; print x ; else print "<= 8"; /if
```

Node:If composite, Next:[For in](#), Previous:[If](#), Up:[Top](#)

If composite

Le construct switch case du C++ ou Java n'existe pas en Scriptol car il est trop limité et inutile (en Scriptol). Il est remplacé par une variante du construct if plus puissante, qui peut tester tout type de variable et plusieurs genres de comparaisons.

La syntaxe est:

```
if expression non-booléenne [:]
  opérateur expression : instructions
  opérateur expression : instructions
  ...
else
  ... instructions ...
/if
```

Une expression non-booléenne est une variable, un littéral, ou toute expression qui ne retourne pas la valeur booléenne vrai ou faux.

On ne place pas le mot-clé "break" à la fin d'un groupe case comme en C. Un break produirait une sortie de la structure de contrôle qui contient le construct if.

Les opérateurs valides sont: =, <, >, <=, >=, !=, <>, in, else.

Le "else" ici est équivalent au "default" du switch case de C.

Exemple:

```
if a
= 1: print 1
= 2: print 2
> 2: print ">2"
else print "<1"
/if
```

```
if x
in array(1,2,3): print "premier"
in array(4,5,6): print "second"
in array(7,8) : print "troisième"
else print "autre"
/if
```

Node:For in, Next:[Scan by](#), Previous:[If composite](#), Up:[Top](#)

For in

Cette structure de contrôle parcourt soit un intervalle d'entiers, soit une tableau et assigne tour à tour chaque élément à une variable.

Syntaxe de for sur un intervalle:

```
for variable in début..fin [step s] [:]
  ... instructions ...
/for
```

Début, fin, et le pas (step) sont des identifiants ou des littéraux.

Le pas est optionnel, la valeur par défaut est 1.

La limite fin est incluse dans l'intervalle, si le symbole d'intervalle est ".." et pas si le symbole est "-".

La fin peut avoir une valeur inférieure au début, si "step" est présent et contient un pas négatif.

La variable conteneur peut être déclarée dans l'en-tête. Dans ce cas elle est locale à la structure de contrôle.

Syntaxe de for sur une liste:

```
for variable in expression-tableau [:]
  ... instructions ...
```

```
/for
```

Dans ce cas, le contenu de l'expression est parcouru et chaque élément assigné à la variable. Cette expression doit être un tableau ou une combinaison produisant un tableau.

Exemples:

```
for i in 1..10
  print i
/for
```

```
for w in montableau
  print w
/for
```

```
for w in arr1 + (arr2 & arr3[0..5])
  print w
/for
```

For à une instruction

La syntaxe courte, d'une instruction est:

```
for variable in liste let instruction-basique
```

Exemples:

```
for w in maliste let print w
for x in 10..1 step -1 let print w + 10
```

Node:Scan by, Next:[Parcourir un tableau à deux dimensions](#), Previous:[For in](#), Up:[Top](#)

Scan by

Permet de scanner des tableaux et d'appliquer une fonction à chaque élément.

Syntaxe:

```
scan a by f
ou
scan a [, b, c ... ]
  ... statements ...
/scan
```

- a, etc... sont des tableaux (array, dict).
- f est le nom d'une fonction.

Le format scan by s'applique à un tableau unique pour la compatibilité avec PHP 5, on peut utiliser plusieurs tableaux avec le format en balises, éventuellement placer l'appel de la fonction dans le bloc. Scan applique la fonction f à chaque élément du tableau, qui lui est donné en paramètre.

Quand le dernier élément du tableau est atteint, le traitement se termine.

Pour que l'élément puisse être modifié par la fonction, le modifieur "alias" doit précéder le type de l'argument dans la déclaration de la fonction utilisée. Scan by requiert une fonction définie par l'utilisateur, et non une méthode de classe.

Sous la forme balisée, chaque élément du tableau est traité par les instructions à l'intérieur du corps du

construct. L'élément courant est accédé par l'indexation vide [].

La syntaxe à une instruction est aussi permise:

```
scan a let instruction
```

Exemples:

```
array a = {1,2,3,4}
void fun(number x) : print x * x; return
scan a by fn

scan a : print a[] * a[]; /scan
```

Ces deux exemples ont le même résultat.
On peut modifier le tableau avec a[] = ...

Exemples:

```
scan a let a[] = 0

scan a
  a[] * a[]
  print a[]
/scan
```

Exemple avec plusieurs tableaux:

```
void mulfon(dyn a, dyn b) print a * b; return
scan a, b by mulfon
ou
scan a, b let print a[] * b[]
```

Node:Parcourir un tableau à deux dimensions, Next:[While](#), Previous:[Scan by](#), Up:[Top](#)

Parcourir un tableau à deux dimensions

Voici un exemple pour créer et parcourir un tableau à deux dimensions:

```
array a = (
  ("a", "b", "c"),
  ("x", "y", "z"),
  (1, 2, 3))
```

Ce tableau contient en fait trois dyn (variables dynamiques) qui contiennent chacune un tableau. Chacun est converti en tableau et assigné à x. Sans la conversion, l'élément serait inséré dans le tableau x.

```
scan a
  array x= a[].toArray()
  scan x
  print x[]
/scan
/scan
```

Il aurait été plus simple d'avoir à éviter la création du tableau temporaire x, ceci peut être fait dans cet exemple (en utilisant la syntaxe à instruction unique):

```
scan a
  scan a[] print a[][]
/scan
```

Ceci fonctionne avec le compilateur natif solc, mais malheureusement, l'interpréteur PHP (4.2) ne le permet pas et donc solp ne peut l'utiliser.

Node:While, Next:[While let](#), Previous:[Parcourir un tableau à deux dimensions](#), Up:[Top](#)

While

La structure de contrôle while est une boucle conditionnelle.

Syntaxe à instruction unique:

```
while expression let instruction
```

Syntaxe standard:

```
while expression [:]
  ... instructions ...
/while
```

Exemple:

```
int x = 10
while x < 20
  print x
  x + 1
/while
```

Une instruction "break" sort de la boucle.

Une instruction "continue" ignore tout ce qui suit et commence une nouvelle boucle.

Node:While let, Next:[Do until](#), Previous:[While](#), Up:[Top](#)

While let

Cette syntaxe est recommandée pour éviter le risque de boucles infinies.

L'instruction qui fait évoluer la valeur de l'expression testée comme condition de l'itération est déplacée après le marqueur /while grâce à l'instruction "let".

Exemple:

```
while x < 10
  if (x mod 2) = 0 continue
  print x
/while let x + 1
```

L'instruction continue saute sur l'instruction let.

Il n'y a pas d'équivalent en C ou PHP parceque l'instruction continue saute le code qui suit, y compris

l'incrémentation de x, d'ou une boucle infinie.

La syntaxe simplifiée et recommandée est:

```
while condition
  ...instructions...
let incrementation
```

Exemple sur une instruction:

```
while x < 10 let x + 1
while x < 10 : print x; let x + 1
```

La condition de la boucle while peut être le mot-clé "forever", et l'on entre dans une boucle infinie, dont on sort par un break.

Node:Do until, Next:[Do case](#), Previous:[While let](#), Up:[Top](#)

Do until

Le bloc d'instructions entre les marqueurs do /do est un nouvel espace de visibilité, et les instructions encloses sont exécutée selon une condition éventuelle.

Syntaxe générale:

```
do
  ... instructions ...
/do [ condition ]
```

et /do condition peut être simplifié en "until".

Le bloc d'instruction est exécuté tant que la condition est fausse, et laissé quand la condition devient vraie.

Syntaxe de do until:

```
do
  .. instructions ...
until expression

do
  print x
  x + 1
until x = 10
```

Node:Do case, Next:[Syntaxe étendue de do](#), Previous:[Do until](#), Up:[Top](#)

Do case

C'est un puissant construct de "pattern-matching". Il contient un ou plusieurs groupes case suivis par un "else" optionnel, et un "always" optionnel. Un seul groupe case est exécuté, le premier dont la condition est satisfaite. Always est toujours exécuté, else quand aucune condition n'est satisfaite.

Syntaxe de do case:

```
do
  case condition : instructions
  [ case condition : instructions ]
  [ else instructions ]
  [ always instructions ]
/do [while expression]
```

- une condition est suivie par un deux-points, ou une fin de ligne. "else" et "always" aussi, pour homogénéiser mais optionnellement. - un seul case est exécuté, ou aucun si l'expression n'est pas reconnue.
- un groupe else équivaut au "default" de de c.
- un groupe always est toujours exécuté, après un autre cas éventuellement. - the end tag is /do or /do forever or /do while.
- Une commande forever ou while fait de la structure un automate. Il peut falloir un break pour en sortir.
- Il ne faut pas placer un break à la fin d'un groupe comme en C. Cela quitterait la structure, non le groupe.

Exemples:

```
do
case nom = "pussycat":    print "c'est un chat"
case nom = "flipper":    print "c'est un dauphin"
case nom = "mobby dick": print "c'est une baleine"
else
  print "un autre animal"
/do
```

```
int state = ON
do
  case state = ON: counter + 1
  case state = OFF: break
  always
    state = getState() ` une fonction quelconque
/do while forever
```

L'automate qui lit son état à partir d'une et continue de tourner jusqu'à ce que l'état OFF soit rencontré, produit par la fonction appelée.

Node: [Syntaxe étendue de do](#), Next: [Break et continue](#), Previous: [Do case](#), Up: [Top](#)

Syntaxe étendue de do

Le marqueur de fin do peut être complété d'une condition, while ou forever.
Le bloc d'instruction sera exécuté au moins une fois et tant que la condition est vraie.

Syntaxe:

```
do
  ... instructions ...
/do while expression
```

```
do
  print x
  x + 1
```

```
/do while x < 3
```

Options - do ... /do ... bloc d'instructions avec ou sans groupes case. - do ... /do while forever ... boucle infinie. - do ... /do forever ... boucle infinie simplifiée. une boucle infinie requiert un "break" pour cesser de s'exécuter.

Node:Break et continue, Next:[Enum](#), Previous:[Syntaxe étendue de do](#), Up:[Top](#)

Break et continue

Commande pour sortir d'une boucle.

Exemple utilisant le mot-clé "forever" (pour toujours) qui crée délibérément une boucle infinie.

```
int x = 0
while forever
  print x
  if x > 100
    break
  /if
  x + 1
/while
```

Quand x atteint la valeur 100, l'instruction break fait que le programme saute au-dela de /while, sur l'instruction après la structure while.

Continue

Cette seconde commande permet de sauter toutes les instructions de la position actuelle, jusqu'en fin de structure, donc de démarrer une nouvelle boucle.

```
int x = -1
while x < 20
  x + 1
  if (x mod 2) = 0 continue
  print x
/while
```

Cet exemple affiche seulement les valeurs paires de x, car lorsque l'on rencontre une valeur impaire, la condition $x \bmod 2$ est vérifiée et une commande continue est exécutée.

Dans cet exemple, j'ai inséré l'incrémentation de x sur la première ligne. Si cela avait été placé après la commande continue, il en résulterait une boucle sans fin puisque l'incrémentation aurait été passée. La syntaxe while .. let évite ceci.

Node:Enum, Next:[Indexation](#), Previous:[Break et continue](#), Up:[Top](#)

Enum

Enum permet d'assigner séquentiellement des valeurs à des identifiants, et en Scriptol, il permet d'assigner des valeurs non entières: des réels ou des textes.

Par défaut un nombre entier est assigné, à partir de zéro, et incrémenté pour chaque identifiant de la

liste.

On utilise un double point pour assigner une valeur donnée à un identifieur et le signe égal pour redémarrer la une valeur donnée.

Syntaxe:

```
enum
  identifieur [ : value | = value ]
  [ identifieur ...]*
/enum
```

```
enum:
  Z,
  UN,
  DEUX,
  TROIS
/enum
```

Cela assigne 0 à Z, 1 à UN, et ainsi de suite. C'est équivalent à:

```
constant int Z = 0
constant int UN = 1
etc...
```

ou:

```
enum: Z:0, UN:1, DEUX:2 /enum
```

Exemple plus complexe assignant des valeur hétérogènes et démarrant une séquence:

```
enum
  Z : "a0",           ... assigne a0
  UN : "a1",         ... assigne a1
  TROIS,             ... assigne 0
  QUATRE : 3.15,     ... assigne 3.15
  CINQ = 5,          ... assigne 5 et redémarre la séquence
  SIX                ... assigne 6
/enum
```

Exemple de syntaxe simplifiée en une instruction:

```
enum ZERO, UN, DEUX, TROIS
```

Node:Indexation, Next:[Intervalle](#), Previous:[Enum](#), Up:[Top](#)

Indexation

La syntaxe d'un indice dans un texte ou un tableau est: [indice]. L'indice doit être une expression simple sans crochets imbriqués. Une expression simple est un nombre littéral, un identifieur, un appel de fonction, ou une expression arithmétique et doit être évaluée comme un nombre entier.

Node:Intervalle, Next:[Intervalles dans une liste](#), Previous:[Indexation](#), Up:[Top](#)

Intervalle

La syntaxe d'un intervalle est:

```
début..fin
```

Pour indiquer une liste, is est enclos entre crochets:

```
nom-liste[début..fin]
```

Début et fin sont des expressions entières. Le dernier élément est inclus dans l'intervalle à moins que l'on utilise l'opérateur: "-"

```
a[0 -- 100]          équivaud à [0..99]
a[x -- y]           équivaud à a[x..y-1]
```

Exemples d'intervalles:

```
0..10
x..y
x * 2 / 4 .. y + 10
```

Vous pouvez utiliser un intervalle pour:

- tester si une valeur est dans un intervalle: if x in 10..100
- scanner un intervalle: for x in 1..100
- extraire une partie de liste: array b = a[x..y]
- changer une partie de liste: a[x..y] = autre-liste

Node:Intervalles dans une liste, Next:[Text](#), Previous:[Intervalle](#), Up:[Top](#)

Intervalles dans une liste

Quand on indice une liste, la limite inférieure ou la limite supérieure de l'intervalle peut être omise.

Il y a trois moyens pour découper une liste:

```
array a = array(x0, x1, x2, x3, x4, x5, x6, x7, x8)
array b = a[..2]
array c = a[3..5]
array d = a[6..]
```

On obtient trois tableaux avec ces contenus:

```
b:  (x0, x1, x2)
c:  (x3, x4, x5)
d:  (x6, x7, x8)
```

On les affiche:

```
b.display()
c.display()
d.display()
```

On doit voir:

```
array (
  [0] => x0
  [1] => x1
  [2] => x2
)
```

et ainsi de suite...

Pour remplacer un intervalle par une autre liste, il suffit d'un assignement:

```
a[3..5] = array("a", "b", "c")
```

Le contenu devient:

```
(x0, x1, x2, "a", "b", "c", x6, x7, x8)
```

Avec la même syntaxe, on remplace une sous-liste par une liste ou un élément:

```
a[3..5] = "xyz"
```

The original liste become:

```
(x0, x1, x2, "xyz", x6, x7, x8)
```

Pour supprimer une sous-liste, on la déclare nil, "not in liste" (pas dans la liste):

```
a[3..5] = nil
```

On a supprimé la sous-liste 3..5 qui est (x3, x4, x5), on obtient:

```
(x0, x1, x2, x6, x7, x8)
```

On peut tester si une valeur appartient à une sous-liste.

Exemple:

```
array a = array("un", "deux", "trois")
if "deux" in a[2..5] print "dedans"
```

Node:Text, Next:[Méthodes de text](#), Previous:[Intervalles dans une liste](#), Up:[Top](#)

Text

Un texte est un objet basique avec des méthodes et qui contient une chaîne de caractères.

Quand un texte est argument d'une fonction, la fonction utilise une copie et non un alias sur le texte original.

Un indice peut être négatif dans un intervalle, pas dans l'indexation.

Syntaxe:

<code>text s</code>	crée un texte.
<code>s = "str"</code>	initialise.
<code>s = s[i]</code>	prend un caractère.
<code>s[i] = s2</code>	remplace un caractère, s2 a un seul caractère.
<code>s = s[i..j]</code>	prend une sous-chaîne, de i jusqu'à j inclus.
<code>s[i..j] = s</code>	remplace une sous-chaîne.
<code>s[i..j] = ""</code>	supprime une sous-chaîne.

Un texte littéral est une chaîne de caractères enclose par des guillemets simples ou doubles.

Le symbole + peut être utilisé avec les textes, et il représente la concaténation de deux chaînes.

Exemple:

```
text b = "préfixe"
text a = b + "suffixe"
```

Le contenu de a sera: "préfixesuffixe".

Méthodes de text

Return Method Fonction
Méthodes de text

Retour	Méthode	Fonction
text	capitalize()	met la première lettre en majuscule.
int	compare(text)	compare lexicographiquement deux textes (ignore maj). retourne -1, 0, 1.
text	dup(int)	retourne un texte dupliqué n fois. Ex: "*"dup(10).
void	fill(text, int)	remplit avec le texte en argument dupliqué n fois.
int	find(text s2)	retourne la position du texte s2 dans le texte, retourne "nil" si non trouvé. (tester si = nil car <> nil ne va pas en PHP)
int	identical(text)	compare, différencie maj/min. Retourne -1, 0, 1
void	insert(int, text)	insère in text à la position donnée.
boolean	isNumber()	retourne true si le texte contient un nombre.
int	len()	retourne la longueur.
int	length()	retourne la longueur.
text	lower()	met en minuscules.
text	ltrim()	supprime blancs et codes de contrôle au début.
text	replace(ft, rt)	remplace chaque occurrence de ft par rt.
void	reserve(int)	alloue la taille donnée pour utiliser comme buffer.
array	split(sep)	découpe le texte en éléments séparés par sep.
int	toInt()	convertit en entier.
natural	toNatural()	convertit en naturel.
real	toReal()	convertit en réel.
text	toText()	cast une chaîne littérale en text (pour C++).
text	trim()	supprime blancs et codes de contrôle en début et fin.
text	rtrim()	supprime blancs et codes de contrôle en fin.
text	upper()	met en majuscules.
text	wrap(int size)	mots non coupés.

Variable dynamique

Sont déclarées avec le type: dyn.

Les variables dynamiques ont les méthodes de tous les autres types de variables mais pas les méthodes de classes déclarées (voir extern). Quand une variable est assignée à un dyn, un cast est requis pour assigner le contenu du dyn à une variable typée.

Méthodes de dyn

- méthodes de conversion: toBoolean, toInt, toNatural, toText, toNumber, toArray, toDict, toFile, toObject.
 - méthodes de test de type: isBoolean, isInt, isInteger, isReal, isNumber, isNatural, isText, isArray, isDict, isFile, isObject.
 - méthode arrayType (retourne le type d'un tableau typé).
-

Node:Séquence et liste, Next:[Tableau](#), Previous:[Variable dynamique](#), Up:[Top](#)

Séquence et liste

Une séquence est une liste soit statique (text) ou dynamique (array ou dict).

Listes et séquences ont les mêmes operateurs, sauf # et | propres aux listes. Les opérations sur les listes sont:

```
[ ] : indice, lecture ou modification de sous-liste.  
+ : fusion de deux séquences. Ou ajoute un élément à une liste.  
- : supprime une séquence d'une autre, ou un élément d'une liste.  
= : comparaison de deux séquences.  
in : teste si un objet est dans une séquence.  
& : intersection de deux listes (donne les éléments communs).  
| : union sans doublons de deux listes.
```

Node:Tableau, Next:[Créer un tableau](#), Previous:[Séquence et liste](#), Up:[Top](#)

Tableau

Il y a deux sortes de listes dynamiques en Scriptol:

```
array: (tableau) indicé par des nombres entiers.  
dict: (dictionnaire) les clés sont des textes.
```

Un tableau est une liste dynamique d'objets ou littéraux indexée.

Un tableau vide est représenté par {}.

Un tableau littéral est une liste d'expressions séparées par une virgule et enclose entre {}.

Une variable peut être un élément de tableau.

Le constructeur débute par le mot-clé "array".

Pour afficher le contenu du tableau a, on type a.display(), et on obtient quelque chose de la forme:

```
array(  
0 : un  
1 : deux  
2 : trois  
)
```

Node:Créer un tableau, Next:[Indexer un tableau](#), Previous:[Tableau](#), Up:[Top](#)

Créer un tableau

Le constructeur d'un tableau a la forme: array(valeur, valeur, etc...)

Le nombre d'argument va de 0 à n. Un initialiseur vide s'écrit: array()

Un tableau littéral s'écrit: { ... valeurs ... } On peut définir un tableau en assignant le constructeur, ou un tableau littéral, ou une valeur unique.

Syntaxe:

<code>array a</code>	crée un tableau.
<code>array a = array()</code>	crée un tableau vide.
<code>array a = {}</code>	crée un tableau vide.
<code>array a = array(x, y, ...)</code>	crée et initialise un tableau.
<code>array a = {x, y, ...}</code>	crée et initialise un tableau.
<code>array a = { 8, 9 }</code>	assigne un tableau littéral.
<code>array a = array(8)</code>	utilise le constructeur.
<code>array a = 3</code>	crée un tableau d'un élément.

Les éléments d'un array peuvent être toute expression, ******* sauf une expression booléenne *******. Si vous placez la valeur vrai dans un tableau PHP retournera vrai chaque fois que vous utilisez l'opérateur "in", avec toute valeur cherchée.

Un tableau peut être déclaré sans assignement de contenu, et rempli par la suite:

```
array a
a.push("un")
a.push("deux")
a.push("trois")
```

Les éléments sont retrouvés par leur position dans le tableau.

Exemples:

```
a[1] = "a"
a[2] = "b"
scan a print a[]
... doit afficher: a b
```

Node: [Indexer un tableau](#), Next: [Itérateur](#), Previous: [Créer un tableau](#), Up: [Top](#)

Indexer un tableau

Les éléments d'un tableau sont accédés par un numéro entier.

Syntaxe:

<code>a[n]</code>	lit l'élément ayant le numéro n.
<code>a[n] = x</code>	remplace l'élément à la position n.
<code>a[n] = nil</code>	efface un élément.
<code>a = {}</code>	efface le tableau entier.
<code>a.[n].upper()</code>	appel d'une méthode sur l'élément n du tableau.

Un index vide désigne l'élément courant:

`a[]` élément à la position courante.

L'indice peut être toute expression qui s'évalue en un nombre entier. L'expression ne peut contenir un élément de tableau.

<code>a[10 + x / 2]</code>	valide.
<code>a[10 + b[5]]</code>	n'est pas valide.

Puisqu'un tableau peut contenir tout type d'objet, ou même d'autres tableaux, il faudra une variable dynamique pour lire un élément, à moins de connaître le type de l'élément à lire.

```
array a = { x, "two", 3 }
dyn x = a[1]
inconnu.
text t = a[2]
int i = a[3]
```

utilisation de dyn pour lire l'élément de type

Quand un élément est ajouté hors de la limite, il est placé en dernière position dans le tableau.
Si vous assignez un élément avec l'instruction suivante, le contenu changera selon le langage cible:

```
a[1000] = "x"
```

En PHP:

```
array(
  0 : un
  1 : deux
  2 : le dernier
  1000: x
)
```

En C++:

```
array(
  0 : un
  1 : deux
  2 : le dernier
  3: x
)
```

En PHP, l'indice 1000 est seulement temporaire, et changera dès qu'une instruction change le tableau. Une instruction de la forme `a[n] = x` sert à modifier la valeur à l'indice donné, il ne faut pas assigner un tableau de cette façon qui ne convient qu'aux dictionnaires (dict), comme on verra plus loin.

Node:Itérateur, Next:[Utiliser un tableau comme pile](#), Previous:[Indexer un tableau](#), Up:[Top](#)

Itérateur

Le contenu d'un tableau peut être parcouru par un itérateur.

On pointe sur le premier élément avec la méthode `begin()`, ou sur le dernier avec `end()`, l'élément courant est obtenu par un index vide de la forme `[]`.

<code>begin()</code>	pointe sur le premier élément et retourne sa valeur.
<code>end()</code>	pointe sur le dernier élément et retourne sa valeur.
<code>inc()</code>	déplace sur l'élément suivant. Retourne la valeur pointée avant, ou nil au-delà du dernier élément.
<code>dec()</code>	déplace sur l'élément précédant. Retourne la valeur ou nil.
<code>index()</code>	retourne l'index de l'élément pointé.
<code>value()</code>	retourne la valeur de l'élément pointé.
<code>[]</code>	indice de l'élément courant.
<code>nil</code>	signifie "not in liste" et est retourné par différentes fonctions quand aucune valeur ne peut être retournée.

Exemple d'utilisation avec le tableau a:

```
a.begin()           ` déplace sur le premier élément
while a[] <> nil    ` teste si la fin de liste est atteinte
  print a[]        ` affiche l'élément actuellement pointé
  a.inc()          ` déplace sur l'élément suivant
/while
```

En ordre descendant:

```
a.end()
while a[] <> nil
  print a[]
  a.dec()
/while
```

Node:Utiliser un tableau comme pile, Next:[Intervalle dans un tableau](#), Previous:[Itérateur](#), Up:[Top](#)

Utiliser un tableau comme pile

Une fois le tableau créé, vous pouvez appliquer différentes fonctions de liste, ou de pile...

```
a.push("item")      ...ajoute un élément à la fin de la liste
a.unshift("item")  ...ajoute un élément au début de la liste
a.pop()            ...lit et enlève le dernier élément
a.shift()          ...lit et enlève le premier élément
```

Vous pouvez ainsi lire et éliminer les éléments d'un tableau par des instructions successives telles que:
print a.shift()

Node:Intervalle dans un tableau, Next:[Opérateurs de tableau](#), Previous:[Utiliser un tableau comme pile](#), Up:[Top](#)

Intervalle dans un tableau

Un intervalle est délimité par un intervalle de positions.

```
a[pos..fin]        la rangée entre "pos" et "fin" inclus.
a[..fin]           du début jusqu'à la position "fin".
a[pos..]           de la position "pos" à la fin du tableau.
a[..]              prend le tableau entier (peu utile).
a[pos..fin] = nil  supprime une rangée d'éléments.
a[pos..fin] = dyn  remplace une rangée par un tableau/élément (id).
```

Node:Opérateurs de tableau, Next:[Tableau à plusieurs dimensions](#), Previous:[Intervalle dans un tableau](#), Up:[Top](#)

Opérateurs de tableau

Un élément, ou un groupe d'éléments, peut être ajouté ou supprimé avec les opérateurs + et -.

Exemple:

```
array a = array("un", "deux")
array b
b = a + array("trois", "quatre")      maintenant le contenu de b est de
quatre éléments.
a = b - array("un", "quatre")         maintenant le contenu de a est:
```

```
("deux", "trois").
```

Seuls les opérateurs arithmétiques + et - sont utilisables sur les tableaux, avec l'opérateur d'appartenance "in".

L'opérateur "in"

Cet opérateur peut être utilisé pour tester si une valeur est contenue dans une liste (array, dict ou même un text), et pour parcourir le contenu également.

Syntaxe et exemples:

```
if variable in array
for variable in array

if x in a : print "dedans"; /if
if x in array(1,2,3) : print "dedans"; /if
for x in a print x
for t in array("un", "deux", "trois") print t
```

Opérateurs binaires de listes dynamiques

Vous pouvez utiliser sur des listes dynamiques (array ou dict) les opérateurs binaires d'intersection et union. L'intersection de deux listes retourne seulement les éléments qui font partie à la fois des deux listes. L'union de deux listes est une nouvelle liste composée des éléments de la première, plus ceux de la seconde moins ceux qui font déjà partie de la première.

```
& intersection
| union
^ complément d'intersection

a = array(1, 2, 3, 4) & array(3, 4, 5, 6)      fait que a contient (3, 4)..
a = array(1, 2, 3, 4) | array(3, 4, 5, 6)     fait que a contient (1, 2, 3, 4, 5, 6).
```

Node:Tableau à plusieurs dimensions, Next:[Contenu d'un tableau](#), Previous:[Opérateurs de tableau](#), Up:[Top](#)

Tableau à plusieurs dimensions

Le nombre de dimensions n'est pas limité. Pour un tableau à deux dimensions, la syntaxe pour...

- accéder à un élément est: `x = nomarray[i][j]`

- changer un élément est: `nomarray[i][j] = x`

Pour créer un élément, la syntaxe est:

```
nomarray[i] = array()
nomarray[i][j] = x
ou
nomarray[i] = array(x)
```

On ne peut créer directement un élément dans un sous-tableau inexistant. L'index i et j supposent qu'il existe déjà i et j éléments.

Node:Contenu d'un tableau, Next:[Dictionnaire](#), Previous:[Tableau à plusieurs dimensions](#), Up:[Top](#)

Contenu d'un tableau en PHP, step by step

Nous devons savoir comment les clés entières d'un tableau en PHP changent selon les opérations que l'on peut accomplir. C'est important pour comprendre les tableaux associatifs et éviter pas mal de bogues.

Après chaque opération le contenu est affiché.

```
array a = {}  
array  
(  
)
```

```
a.push("un")  
array  
(  
  [0]=> un  
)
```

```
a + array("deux", "trois", "quatre")  
array  
(  
  [0]=> un  
  [1]=> deux  
  [2]=> trois  
  [3]=> quatre  
)
```

```
a.shift()           ... le premier élément est supprimé et les clés sont renumérotées.  
array  
(  
  [0]=> deux  
  [1]=> trois  
  [2]=> quatre  
)
```

```
a[1000] = "mille"  
array  
(  
  [0]=> deux  
  [1]=> trois  
  [2]=> quatre  
  [1000]=> mille  
)
```

```
a.unshift("x")     ... toutes les clés sont renumérotées, même la clé 1000.  
array  
(  
  [0]=> x  
  [1]=> deux  
  [2]=> trois  
  [3]=> quatre  
  [4]=> mille  
)
```

Créons deux nouveaux tableaux:

```
array a = array("un", "deux")  
array b = array()  
b[1000] = "mille"  
a + b  
Les clés sont renumérotées.  
array  
(  
  [0]=> un  
  [1]=> deux  
  [2]=> mille  
)
```

Si on remplace `a + b` par `a.push("mille")` le résultat sera le même.

Node:Dictionnaire, Next:[Créer un dictionnaire](#), Previous:[Contenu d'un tableau](#), Up:[Top](#)

Dictionnaire

Un dict est une liste dynamique de couples clé et valeur. Les clés sont toujours des textes. Les valeurs peuvent être tout objet.

La clé et la valeur peuvent être des variables. Le format pour un couple clé et valeur est: clé:valeur. (L'équivalent PHP est: clé => valeur). Un dict vide est représenté par `{}`. Un dict littéral est une liste de couples séparés par une virgule, et enclos entre `{}`t.

Au contraire du tableau, le dict est rempli avec des assignements:

```
d[k] = "b"
d["un"] = "element 1"
```

Array et dict ont les mêmes méthodes, mais certaines sont plus utiles avec l'un ou l'autre type.

Node:Créer un dictionnaire, Next:[Indexer un dict](#), Previous:[Dictionnaire](#), Up:[Top](#)

Créer un dictionnaire

Un dict peut être créé à partir d'un littéral ou un constructeur.

Syntaxe:

```
dict d                                crée un dict.
dict d = {x:v, y:w,...}              crée et initialise un dict.
dict d = dict(x:v, y:w,...)          c crée et initialise un dict.
```

Les valeurs peuvent être tout type d'objet.

La clé peut être une variable et la valeur une expression.

Exemple:

```
text k = "a"
text v = "b"
dict d = dict(k : v)
dict d = dict(k : v + "x".dup(4) + 20.toText())
```

Cet exemple enregistre "bxxxx20" dans le dict d, avec la clé "a".

Node:Indexer un dict, Next:[Intervalle dans un dict](#), Previous:[Créer un dictionnaire](#), Up:[Top](#)

Indexer un dict

Les valeurs dans un dict sont accédées par une clé textuelle.

Syntaxe:

<code>d["clé"]</code>	prend le premier élément avec la clé "clé".
<code>d[clé] = valeur</code>	remplace une valeur ou ajoute un couple clé-valeur si la clé n'est pas déjà dans le dict.
<code>d[clé] = nil</code>	supprime un élément.
<code>d = {}</code>	efface le dict.

Exemple:

```
dict d
d["program"] = "ce que l'on veut accélérer"
print d["program"]           affiche le texte ci-dessus.
```

Node: [Intervalle dans un dict](#), Next: [Méthodes de array et dict](#), Previous: [Indexer un dict](#), Up: [Top](#)

Intervalle dans un dict

La façon normale d'utiliser un dictionnaire est par le moyen des clés ou itérateurs. En quelques occasions il peut être utile d'accéder à un intervalle de valeurs directement.

- Quand on ajoute un élément ou un autre dict à un dict, au moyen d'intervalle, de push, unshift, PHP génère une nouvelle clé pour cet élément. La nouvelle clé est un nombre.
- Si vous remplacez un intervalle par un autre dict, certains éléments peuvent être perdus. Cela se passe aussi lors de la fusion.
- Les clés du dict qui remplace un intervalle ne sont pas conservées.

Exemples d'affichage:

Il doivent présenter toutes les clés et valeurs dans le dict.

```
dict d = {"a":"alia", "b":"beatrix", "c":"claudia"}           création du dict
d.display()                                                  affichage
for k,v in d : print k, v; /for                               affichage avec une boucle
for
number i = 0
d.begin()
while i < d.size()                                           affichage avec un itérateur
  print i,"i", d.clé(), d[]
  d.inc()
/while let i + 1
```

Exemple: lire le dernier couple:

```
v,k = d.end(), d.key()
```

L'ordre est important car `d.end()` déplace le pointeur à la fin.

Node: [Méthodes de array et dict](#), Next: [Tableaux typés](#), Previous: [Intervalle dans un dict](#), Up: [Top](#)

Méthodes de array et dict

Return	Name	Action
dyn	<code>begin()</code>	pointe sur le premier élément.
dyn	<code>dec()</code>	retourne un élément et décrémente le pointeur.
void	<code>display()</code>	affiche le tableau.
dyn	<code>end()</code>	pointe sur le dernier élément.
boolean	<code>empty()</code>	return true si le tableau est vide.
int	<code>find(dyn)</code>	recherche un élément, retourne l'index ou nil.
void	<code>flip()</code>	les valeurs deviennent clés et inversement.
dyn	<code>inc()</code>	retourne un élément et incrémente le pointeur.
int	<code>index()</code>	retourne l'index de l'élément pointé.
void	<code>insert(int, dyn)</code>	insère un élément à un index entier (pas un text).
text	<code>join(text sep)</code>	convertit en text avec le séparateur donné.
text	<code>key()</code>	retourne la clé de l'élément pointé.
void	<code>kSort()</code>	réordonne les index en préservant les associations.
boolean	<code>load(text nom)</code>	charge un fichier dans un tableau ("file" de PHP).
dyn	<code>min()</code>	retourne la valeur la plus faible.
dyn	<code>max()</code>	retourne la plus forte valeur.
void	<code>pack()</code>	rend les éléments du tableau consécutifs.
dyn	<code>pop()</code>	lit et supprime le dernier élément.
dyn	<code>pop(int)</code>	lit et supprime l'élément à la position donnée.
void	<code>push(val)</code>	ajoute un élément à la fin.
dyn	<code>rand()</code>	retourne un élément à une position aléatoire.
array	<code>reverse()</code>	retourne la liste en ordre inversé.
dyn	<code>shift()</code>	lit et supprime le premier élément.
int	<code>size()</code>	retourne le nombre d'éléments.
void	<code>sort()</code>	trie les valeurs en ordre ascendant.
void	<code>store(text nom)</code>	sauve le tableau dans un fichier.
number	<code>sum()</code>	calcule la somme des éléments.
void	<code>unique()</code>	supprime les valeurs en double, premier gardé.
void	<code>unshift(dyn)</code>	insère un élément en première position.
dyn	<code>value()</code>	retourne la valeur pointée. <code>k,v = d.key(), d.value()</code> lit le couple clé:valeur

Node:Tableaux typés, Next:[Constructeur et littéral](#), Previous:[Méthodes de array et dict](#), Up:[Top](#)

Tableaux typés

Un tableau typé contient un type d'objets unique. Il est beaucoup plus efficient que le tableau commun parcequ'il est indexé, et non associatif, et contient directement des objets de 32 ou 64 bits ou des pointeurs, plutôt que des objets dynamiques (dyn).

Un tableau d'entiers est 100 fois plus rapide qu'un tableau standard pour les exécutable binaires, mais il n'y a pas de différence en PHP.

Les tableaux typés sont aussi dynamiques, la taille n'est ni prédéfinie, ni limitée.

Node:Constructeur et littéral, Next:[Tableau d'entiers](#), Previous:[Tableaux typés](#), Up:[Top](#)

Constructeur et littéral

Le constucteur d'un tableau typé a la forme:

```
type(...éléments...)
```

Exemple:

```
int(1, 2, 3)
```

Le nombre d'élément va de 0 à n.

Quand il y a un seul élément, il n'y a pas de différence entre un constructeur de tableau typé et un constructeur scalaire:

```
int(1)
text("text")
```

Ce n'est pas un problème lors de l'assignement, puisque l'on peut créer un tableau à un seul élément à partir d'un scalaire, mais il y a ambiguïté dans une expression.

Aussi, dans une expression on doit utiliser un tableau littéral:

```
type{...éléments...}
```

Les accolades ont la signification "tableau de".

Node:Tableau d'entiers, Next:[Tableau de textes](#), Previous:[Constructeur et littéral](#), Up:[Top](#)

Tableau d'entiers

La syntaxe pour déclarer un tableau d'entiers est:

```
int[] i = int(x, y, ....)
```

Une fois qu'il est déclaré, il est utilisé comme un tableau mixte, mais seuls les nombres entiers peuvent être ajoutés au tableau.

Lors de la création on peut assigner un constructeur ou un littéral.

Un constructeur à un seul élément équivaut à un constructeur d'entier. Dans une expression il faudra utiliser un littéral.

Exemples:

```
int[] x = int(5)
int[] x = int(1,2)
int[] x = y + int{5}
```

Node:Tableau de textes, Next:[Tableau de number](#), Previous:[Tableau d'entiers](#), Up:[Top](#)

Tableau de textes

La déclaration est:

```
text[] t = text(a, b, "demo", ...)
```

Les règles sont les mêmes que pour les entiers.

Node:Tableau de number, Next:[Tableau d'objets](#), Previous:[Tableau de textes](#), Up:[Top](#)

Tableau de real, natural, number

Les déclarations sont:

```
real[] r    = real(1.0, 2.0, etc...)
natural[] n = natural(1, 2, etc...)
number[] n  = number(1, 2.0, ...)
```

Node:Tableau d'objets, Next:[Assignement et conversion](#), Previous:[Tableau de number](#), Up:[Top](#)

Tableau d'objets

La déclaration est:

```
class uneclasse
...
/class

uneclasse objet1          ... instance
uneclasse objet2          ... instance
objet[] = objet(objet1, objet2, ...)    ... tableau d'objets
```

Node:Assignement et conversion, Next:[Tableau typé et dyn](#), Previous:[Tableau d'objets](#), Up:[Top](#)

Assignement et conversion

Vous pouvez assigner un tableau typé à un tableau mixte:

Exemple:

```
int[] i = int(...)
array a = i
```

Mais vous ne pouvez créer un tableau mixte avec un constructeur typé:

Exemples:

```
{1,2,3}          ... c'est un tableau mixte, avec des éléments entiers
int[] i = {1,2,3} ... NON VALIDE, il faut plutôt écrire:
int[] i = int{1,2,3} ... valide.
```

Exemples:

```
array a = int(1,2)          ... NON valide
array a = array(int(1,2))   ... NON valide
```

Node:Tableau typé et dyn, Next:[Assignement direct](#), Previous:[Assignement et conversion](#), Up:[Top](#)

Tableau typé et dyn

On peut assigner un tableau typé à un dyn.

```
int[] i = int(1,2)
dyn d = dyn(i)
```

Deux méthodes sont fournies pour utiliser ce dyn:

arrayType() retourne le type de tableau contenu dans le dyn.

toList(int = 0) retourne le tableau typé contenu.

Le paramètre est le type, et est requis quand le dyn se contient tableau mixte, que l'on veut convertir en tableau typé.

Types reconnus:

```
$TYPE_INT          tableau d'entiers
$TYPE_TEXT
$TYPE_REAL
$TYPE_NATURAL
$TYPE_NUMBER
$TYPE_OBJECT
```

Exemple:

```
d = dyn(int(1,2))
if d.arrayType()
= $TYPE_INT:
    int[] i = d.toList()
= $TYPE_TEXT:
    text[] t = d.toList()
/if
```

Node:Assignement direct, Next:[Limitations et compatibilité](#), Previous:[Tableau typé et dyn](#), Up:[Top](#)

Assignement direct

Pour la compatibilité avec PHP, ajouter un élément à un tableau au-dela de la taille actuelle, a le même effet qu'ajouter cet élément, quelque soit l'index donné.

Par exemple:

```
int[] i = nil
i[10] = 10                                équivaut à: i.push(10)
```

Empiler des valeurs est le moyen correct pour remplir un tableau.

Si on veut réellement placer un élément à une position donnée, on doit remplir le tableau avec des valeurs nulles...

```
for int k in 0 .. 10 let i.push(0)
```

Vous pouvez maintenant placer un élément en position 10.

Node:Limitations et compatibilité, Next:[File](#), Previous:[Assignement direct](#), Up:[Top](#)

Limitations et compatibilité

- On ne peut appliquer une méthode directement à un élément indicé.

```
i[10].toText()    .. non valide
int x = i[10]
x.toText()       .. valide
```

- Vous ne pouvez comparer directement un élément et une chaîne. Ex:

```
text[] t = text("un", "deux")
if t[0] = "one"    .. non valide
text t2 = t[0]
if t2 = "one"     .. valide
```

- Avec PHP 4.3, insérer un objet dans un tableau fait que les valeurs sont insérées, pas le tableau lui-même! Je considère que c'est une boguee tdonc vous ne pouvez insérer directement des objets, mais vous pouvez insérer un tableau d'objet.

- La fonction "array_diff" de PHP ne fonctionne pas quand le tableau contient des objets, on ne peut donc soustraire de tels tableaux.

Node:File, Next:[Read](#), Previous:[Limitations et compatibilité](#), Up:[Top](#)

File

File est un objet virtuel, pour traiter des fichiers locaux ou distant.

Pour une explication complète du système de fichier, voir "fopen" dans le manuel C ou PHP.

Un objet file est créé par la déclaration d'une instance de type file. Le fichier sur le disque est créé par la commande open. Elle permet d'accéder à un fichier en différents modes, selon le second paramètre de la méthode.

La structure de contrôle error permet de tester si le fichier a été ouvert correctement ou non.

Syntaxe pour créer ou ouvrir un fichier:

```
file nomfic          declare un file.
nomfic.open(path, mode) ouvre un fichier selon un chemin et un mode.
```

Types de chemins:

```
http://path          fichier distant http.
ftp://path           fichier distant ftp.
path                 fichier local.
```

Modes:

```
"r"                  lecture seulement, à partir du début.
"w"                  écriture seulement à partir du début.
"r+"                 lecture ou écriture au début du fichier.
"a"                  ajout en fin de fichier en écriture seule.
"a+"                 lecture à la position courante, ou écriture à la fin.
```

Les méthodes de fichier sont:

```
return             méthode                action
int eof()          retourne true quand la fin de fichier est
atteinte.
```

int	open(text, text)	crée ou ouvre un fichier. Positionne le drapeau
"error".		
int	close()	ferme le fichier.
text	read(int)	lit un bloc selon la taille en argument, retourne
un text.		
text	readLine()	lit et retourne la ligne suivante dans un fichier
de texte.		
int	seek(int)	va à la position donnée en argument, retourne 1
si ok.		
int	size()	retourne la taille du fichier.
int	time()	retourne la date de dernière modification.
int	write(text)	écrit le texte donné en argument, retourne la
taille écrite.		
void	writeLine(text)	écrit le texte en argument.

Exemples:

file nomfic	crée un objet file.
nomfic.open("monfichier", "r")	ouvre un fichier physique.
nomfic.close()	ferme le file.
boolean b = nomfic.eof()	assigne vrai à b si la fin de fichier est atteint.
nomfic.seek(250)	lecture ou écriture en sautant 250 octets.

Les fichiers distants ne sont pas gérés par Scriptol C++ et l'interpréteur pour l'instant.

Node:Read, Next:[Write](#), Previous:[File](#), Up:[Top](#)

Read

Un fichier peut être lu ligne par ligne, ou par blocks binaires.

Dans le premier cas, la méthode readLine() est utilisée, sinon, on utilise la méthode read avec la taille du bloc en paramètre.

Le code de fin de ligne dans le fichier est inclus, et peut être supprimé par la méthode rTrim().

Exemples:

text t = nomfic.readLine()	lecture d'une ligne de texte.
text t = nomfic.read(1000)	lecture de 1000 octets.

Node:Write, Next:[Dir](#), Previous:[Read](#), Up:[Top](#)

Write

La méthode write place le contenu d'une variable de texte dans un fichier. Une fois le fichier ouvert soit avec le mode "w" ou "a", et donc enregistré soit à partir de zéro ou à la fin de son contenu actuel, chaque nouvel appel à write ajoute un nouveau texte à la fin.

Exemple simple:

text nomfic = "monfic"	` nom du fichier
file sortie	` nom de l'objet virtuel
sortie.open(nomfic, "w")	` ouvert en écriture
error? die("impossible d'ouvrir " + nomfic)	
for text ligne in maliste	` un tableau quelconque

```
    sortie.write(ligne)           ` enregistrer un élément
/for
sortie.close()                  ` fermer le fichier
```

Node:Dir, Next:[La structure de contrôle error](#), Previous:[Write](#), Up:[Top](#)

Dir

Le type dir a pour méthodes le constructeur et get(), et sert à utiliser les fonctions intégrées de gestion de répertoire.

Le constructeur sert à créer un dir ou à l'afficher.

```
void dir(text)                  constructeur. L'argument est un chemin.
array get()                     retourne le contenu du répertoire dans un tableau.
```

```
print dir("c:/jeux")
```

ou

```
dir x = dir("sousdir")
print x
```

ou

```
array a = x.get()
a.display()
```

Le séparateur de répertoires est "/" pour tous les systèmes.

Les fonctions de gestion de répertoires sont:

```
dir opendir(text)              ouvre un répertoire, retourne un dir.
text readdir(dir)              lit l'entrée suivante et retourne le nom.
void closedir(dir)             ferme le répertoire.
void rewinddir(dir)            revient à la première entrée.
boolean is_dir(text)           retourne vrai si l'entrée est un répertoire.
boolean is_file(text)          retourne vrai si l'entrée est un fichier.
```

Les fonctions chdir, mkdir, rmdir sont détaillés dans le chapitre sur les fonction intégrées.

Exemple d'utilisation qui affiche le contenu d'un répertoire:

```
dir monrep = opendir("/chemin")
while forever
    text t = readdir(monrep)
    if t = nil break
    if t[0] <> "." print t
/while
```

Node:La structure de contrôle error, Next:[Scopes](#), Previous:[Dir](#), Up:[Top](#)

La structure de contrôle error

Après une instruction open, le construct error ... /error devrait être exécuté en cas d'erreur d'accès. Mais l'interpréteur peut stopper le programme avant d'atteindre le construct. Sinon le contenu du bloc sera exécuté en cas d'erreur.

La syntaxe est:

```
xxx.open("nom fichier")
error
  ...instructions...
/error
ou:
  xxx.open("nom fichier"); error ? action
ou:
  xxx.open("nom fichier"); error action
```

Exemple:

```
file monfic
monfic.open("nom", "r")
error? exit()
```

Si le fichier n'est pas trouvé ou ne peut être ouvert, le programme s'arrête.

Node:Scopes, Next:[Extern](#), Previous:[La structure de contrôle error](#), Up:[Top](#)

Scopes

Les règles de visibilité sont celles des principaux langages procéduraux, non celles des langages de scripts.

Scriptol ajoute des règles de sûreté, il ne permet pas de donner le même nom à deux objets dans les scopes imbriqués (par exemple, dans le scope global et celui d'une fonction). Mais les mêmes noms peuvent se réutiliser dans des scopes successifs.

Il y a quatre niveaux de visibilité: - global, - classe, - corps d'une fonction, - et le corps d'une structure de contrôle.

La visibilité d'une variable est le niveau dans lequel elle est déclarée: global, classe, fonction ou bloc délimité (if, do, for, etc.), ainsi que les niveaux contenus.

On ne peut redéclarer une variable là où elle reste visible. On ne peut la redéclarer que quand son scope est clos, dans des fonctions successives ou des structures de contrôles successives.

L'en-tête du construct for fait partie du scope du bloc.

```
for text t in a          t est visible seulement dans la boucle for.
  t = a.inc()
/for
```

La partie let qui termine une boucle while est dans le scope de la boucle.

Les arguments d'une fonction sont dans le scope de la fonction.

A l'intérieur d'une fonction, si une variable est référencée avant tout assignement, elle référence une

variable globale si elle existe, sinon c'est une erreur.

A l'intérieur d'une méthode d'une classe, si elle est référencée avant tout assignement, elle référence un attribut s'il existe, sinon c'est une erreur.

Les variables globales ne sont pas visibles dans une classe, sauf les variables externes.

Les variables externes (celles de PHP, Apache, etc...) sont toujours dans le scope, puisqu'il n'y a pas de contrôle sur elles. Donc vous devez connaître leur nom pour ne pas le réutiliser, car les variables Scriptol sont transformée en variable PHP avec \$ devant.

Si vous utilisez des variables externes dans une fonction, il faut les déclarer "global", car le compilateur ne les gère pas.

Node:Extern, Next:[Variables et constantes](#), Previous:[Scopes](#), Up:[Top](#)

Extern, externe variables, constantes et fonctions

Les fonctions, variables et constantes PHP ou C++ peuvent être utilisées dans du code Scriptol, comme des objets Scriptol.

Leur visibilité est externe, donc elles sont visibles comme les objets globaux on de classes.

Node:Variables et constantes, Next:[Fonctions](#), Previous:[Extern](#), Up:[Top](#)

Variables et constantes

Les variables et constantes de PHP peuvent être utilisées comme celles de Scriptol dès lors qu'elle sont déclarées avec le mot-clé "extern".

Les déclarations externes comme les includes sont placées en tête de fichier.

La syntaxe pour une variable externe est:

```
extern type identifieur
extern constant type identifieur
```

Aucun test de contrôle sur le type ou le scope d'un objet externe. Dans une fonction il faut une déclaration "global".

Exemple:

```
global argv
array a = $argv
```

Une constante PHP est déclarée comme une variable avec le mot-clé "constant":

```
extern constant text PHP_VERSION
...
print PHP_VERSION
```

Il est possible aussi d'utiliser une variable PHP ou C++ sans déclaration avec le préfixe \$ suivie d'un identifieur.

Celui-ci peut commencer par un caractère de soulignement.

<code>\$var_name</code>	
<code>\$(constant_name)</code>	in PHP
<code>\$constant_name.</code>	in C++

Node:Fonctions, Next:[Classe externe](#), Previous:[Variables et constantes](#), Up:[Top](#)

Fonctions externes

Les fonctions PHP peuvent être utilisées sans déclaration, ce qui n'est pas le cas des fonctions C++. On peut aussi les déclarer avec le mot-clé `extern`. Une valeur par défaut est spécifiée par un assignement. Cette valeur est utilisée en C++, elle est ignorée par PHP.

Syntaxe et exemple:

```
extern type identifieur(... arguments...)

extern text substr(text, int, int = 0)
```

Node:Classe externe, Next:[Types externes](#), Previous:[Fonctions](#), Up:[Top](#)

Classe externe

Pour utiliser les méthodes de classes PHP ou C++, vous devez déclarer ces classes et les membres que vous voulez utiliser.

Exemple:

```
extern
  class phpClass
    int phpAttribute
    void phpMethod(int argument)
    ... déclaration ...
  /class
/extern
```

Node:Types externes, Next:[Insérer du code cible](#), Previous:[Classe externe](#), Up:[Top](#)

Types externes

C++ permet de définir des types avec les directives `"typedef"` and `"#define"`. Il est possible d'intégrer ces nouveaux types avec l'instruction `Scriptol define`.

Exemple: `define uint`

Cela correspond à `"#define uint unsigned int"` en C++.

Aucun code n'est généré par l'instruction `define`: le code C++ doit se trouver dans un fichier C++ à inclure. Voir "L'instruction `define`" pour plus de détails.

Node:Insérer du code cible, Next:[Class](#), Previous:[Types externes](#), Up:[Top](#)

Insérer du code cible

Si vous voulez insérer directement du code PHP dans votre programme, utilisez les symboles `~~` pour définir le début et la fin du code à inclure.

Le compilateur ignore le contenu, comme s'il s'agissait de commentaires mais l'inclut dans le code généré, PHP ou C++.

Vous pouvez insérer du code spécifique PHP ou C++ dans un même programme.

- Pour insérer du code PHP, utiliser:

```
require_once("lecode.php")
```

Cette instruction n'a d'effet que pour l'interpréteur PHP, compilée en C++ elle est ignorée.

- Pour insérer du code C++, utiliser:

```
#include "lecode.c++"
```

Ce sera compilé en C++, mais pour l'interpréteur PHP c'est un commentaire.

Node:Class, Next:[Définir une classe](#), Previous:[Insérer du code cible](#), Up:[Top](#)

Class

Une classe est une structure qui contient des variables (attributs) et a des fonctions (méthodes).

Une fois qu'une classe est définie, elle devient un nouveau type complexe du langage avec lequel on déclare des objets, instances de la classe. On utilise les attributs et méthodes de l'objet avec une commande de la forme:

```
x = objet.attribut
objet.methode()
```

Node:Définir une classe, Next:[Constructeur et instance](#), Previous:[Class](#), Up:[Top](#)

Définir une classe

La description d'une classe débute avec le mot-clé "class" et se termine par "/class".

Les attributs (variables) doivent être placés avant les fonctions. Une classe ouvre un espace de visibilité. Les variables globales et les fonctions définies ne sont pas visibles à l'intérieur d'une classe. Les attributs et méthodes de la classe ou d'une instance d'une autre classe sont visibles dans toutes les méthodes de la classe avec le nom de l'instance en préfixe.

Exemple d'une déclaration de classe:

```
class voiture
    ...membres...
/class
```

Exemple de classe avec attributs et méthodes:

```
class voiture
    int lavitesse = 50
    int passagers = 4

    void voiture.vitesse(int i)
        int x = lavitesse * 2
```

```
    return x
/class
```

Exemple d'instance avec référence aux méthodes:

```
voiture mavoiture
print mavoiture.passagers
print mavoiture.vitesse()
```

Node:Constructeur et instance, Next:[Méthodes et attributs statiques](#), Previous:[Définir une classe](#),
Up:[Top](#)

Constructeur et instance

Un constructeur est une méthode dont le nom est celui de la classe et qui est appelée lors de la création d'une instance de la classe.

Il a toujours le type de retour "void", qui signifie: rien.

Exemple de constructeur:

```
class Voiture
    int vitesse

    void Voiture(int p)           ` cette méthode est un constructeur
                                ` vitesse est initialisée lors de la déclaration
d'une instance
    return
/class
```

La syntaxe pour créer une instance est:

Nomclasse nominstance = Nomclasse(argument [, arguments])

ou:

Nomclasse nominstance pour un constructeur sans arguments

Exemple:

```
Voiture mavoiture = Voiture(60)           ... crée une voiture avec une vitesse
de 60
Camion moncamion                               ... équivalent à ci-dessous
Camion moncamion = Camion()
```

Node:Méthodes et attributs statiques, Next:[Héritage](#), Previous:[Constructeur et instance](#), Up:[Top](#)

Méthodes et attributs statiques

Il peut être pratique de rassembler toutes les fonctions concernant une tâche dans une classe et les déclarer "static". Ces fonctions peuvent alors être appelées directement avec le nom de la classe, sans avoir à créer d'instance.

Exemple:

```
node, ext = Dir.splitExt(chemin)
```

Les attributs peuvent également être statiques. Un attribut statique est commun à toutes les instances

et celles qui sont héritées. Une méthode statique ne peut référencer des attributs, puisqu'ils n'existent que dans les instances de la classe sauf s'ils sont statiques aussi.

On ne peut appeler une autre méthode dans le corps d'une méthode statique..

Le mot-clé "static" doit précéder le type de l'objet.

Un message d'erreur est émis quand une fonction statique utilise un attribut non statique ou qu'elle appelle une autre méthode.

Exemple d'attributs et méthodes statiques:

```
class Voiture
  static usine = "Xxx"
  static void denomme(text x)
    usine = x
  return
/class
```

```
Voiture.denomme("nom")
Voiture.usine = "nom"
```

Attributs et méthodes statiques peuvent être associés à une instance également:

```
Voiture maVoiture
maVoiture.denomme("nom")
maVoiture.usine = "nom"
```

L'effet sera le même.

Node:Héritage, Next:[Surcharge](#), Previous:[Méthodes et attributs statiques](#), Up:[Top](#)

Héritage

Une classe peut hériter des attributs et méthodes d'une autre classe, si elle est déclarée sous-classe de celle-ci.

La classe "nom" hérite des attributs et méthodes de "autrenom". Cela fonctionne aussi avec les attributs et méthodes statiques.

La syntaxe de la déclaration d'héritage est:

```
class nom is autrenom
```

Exemple:

```
class Vehicule
  int fuel
  void Vehicule()
    fuel = 100
  return
/class

class Voiture is Vehicule
  int passagers
  void Voiture()
    passagers = 3
    fuel = 50
  return
/class
```

```

class Camion is Vehicule
  int charge
  void Camion()
    fuel = 200
    charge = 1000
  return
/class

Camion bibendum
Voiture daisy
print bibendum.charge      attribut de la classe Camion
print bibendum.fuel        attribut de la superclasse Vehicule
print bibendum.passagers   mauvais! passagers non accessible à Camion!
print daisy.passagers       bon, passagers est attribut de Voiture

```

Node:[Surcharge](#), Next:[XML statique](#), Previous:[Héritage](#), Up:[Top](#)

Surcharge

Une méthode peut être redéfinie avec des arguments différents, à la fois dans la même classe et dans les classes dérivées. Le type de retour de toutes les versions de la méthode doit être le même.

Exemple:

```

void ajoute(int x, int y)
void ajoute(real x, natural y)

```

Vous avez juste à appeler la méthode "ajoute" avec le type d'arguments voulus, le compilateur associe la définition correspondante...

Le langage cible C++ requiert que les méthodes surchargées aient le même type de retour, cela est conservé en Scriptol.

Node:[XML statique](#), Next:[XML light](#), Previous:[Surcharge](#), Up:[Top](#)

XML statique

NOTE: XML statique est implémenté dans l'interpréteur uniquement pour le moment.

Le compilateur Scriptol en binaire, utilise la class dom, avec des méthodes similaires, et requiert le fichier libdom.sol, qui décrit la classe.

Le compilateur Scriptol en PHP ne reconnaît pas XML pour le moment.

A l'avenir le présent chapitre s'appliquera également aux compilateurs.

XML est une structure de données en Scriptol. Un document XML est défini sous forme light, et ensuite utilisé pour stocker et délivrer des données, juste comme une classe, avec plus de flexibilité.

Le document XML peut être utilisé statiquement, seul le contenu des éléments et les valeurs des attributs sont modifiés.

On accède à un élément par la liste chaînée des noms d'élément le contenant.

```

dom.document.client.getData()

```

Des méthodes permettent aussi de changer la structure, ajouter ou supprimer des éléments. Ceci est couvert dans la section "XML dynamique".

Node:XML light, Next:[Définir un document XML](#), Previous:[XML statique](#), Up:[Top](#)

XML light

Un document XML peut être inclus dans un source scriptol sous forme "légère", sans les crochets angulaires.

La syntaxe d'une structure XML est:

```
xml [name]          ... un nom optionel pour le document.
  [xul]             ... le modifieur xul optionel.
  [headers]        ... des éléments d'en-tête optionels.
  [doctype]        ... des "doctype".
  [include]        ... des instructions include si besoin pour xul.
  tag              ... un ou plusieurs éléments balisés.
/xml
```

Include

La directive include permet d'inclure un autre document xul.

Syntaxe:

```
include href = "other-xul-document-location"
```

En-tête

Syntaxe:

```
<? identifieurs et attributs ?>
```

Doctype

Syntaxe:

```
<! identifieurs et attributs >
```

Noter que cela finit par > et non !>

Element

Un élément balisé a la forme:

```
nom [attribut [[,] attribut]* ]      une liste d'attributs optionels
  [nom2]*                            aucun, un ou plusieurs sous-éléments.
  text                                ou un texte.
  ...
/ nom                                le nom de l'élément ici est optionel;
```

Si la liste des attributs continue sur la ligne suivante, une virgule doit être au moins à la fin de la ligne.

Définir un document XML

Un document XML se déclare comme une classe, enclos entre les marqueurs XML et /XML, au format light.

```
xml nomdocument
  nomelement [attributs]
    "données"
  ou
  ~~
  données sur plusieurs lignes
  ~~
  ou
  éléments imbriqués
/ nomelement
/xml
```

On déclare ensuite des instances XML.

```
nomdocument instance1
nomdocument instance2
```

Les instances sont créées lors de la compilation, les changements sur le document lors de l'exécution ne se retrouvent pas dans les instances.

Chargement et sauvegarde

Des documents externes sont chargés dans un programme Scriptol avec la méthode "load", et un nom de fichier ou chemin en paramètre.

```
xml x
/xml
x.load("demo")
```

ou :

```
x xinstance
xinstance.load("demo")
```

Le document, chargé et complété, ou défini dans un source Scriptol, est sauvé avec la méthode "save" et un nom de fichier ou chemin en paramètre.

Assignement XML

Vous pouvez assigner à une instance XML un document. Il faut pour cela déclarer un document maître XML vide, servant à déclarer l'instance, avant de l'assigner.

Exemple utilisant "nomdocument" défini ci-dessus.

```
xml dom
/xml

dom instance1 = nomdocument
```

Le contenu de "instance1" est maintenant celui du document "nomdocument".
Vous pouvez créer autant d'instances de "dom" que vous voulez.

Node:Itérateur XML, Next:[Utiliser les données](#), Previous:[Assignement XML](#), Up:[Top](#)

Itérateur XML

L'itérateur est un ensemble de méthodes qui peuvent être utilisées avec les méthodes de balise. Les méthodes de balises sont associées au nom du document ou d'instance de document, mais concernent la balise actuellement pointée.

Pour parcourir le contenu d'un document ou d'une instance, on utilise un itérateur, qui se compose de plusieurs méthodes:

```
reset()          aller au premier élément du document.
begin()          aller au premier élément du niveau courant.
inc()            pointer sur l'élément suivant.
found()          retourne vrai si un élément est trouvé par la méthode inc()
ou down().
down()          pointer sur l'élément contenu.
up()            revenir à l'élément contenant.
```

Le document peut être parcouré en partant de la fin avec ces méthodes:

```
end()           aller au dernier élément du même niveau.
dec()           pointer sur l'élément précédent.
```

Exemple d'utilisation d'itérateur:

```
demo.reset()
demo.down()
while demo.found()
    print demo.getData()
let demo.inc()
```

L'itérateur agit sur un niveau du document. Pour atteindre un niveau donné de contenu, on utilise la méthode `at()` avec la liste chaînée des noms de niveaux contenant.

```
xml demo
  document
    element
      "data'"
    /
    element
      "data"
    /
  /
/xml

demo.document.element.at()
```

Nous sommes au niveau de "element" contenu dans document. La méthode `reset()` ramène l'élément

courant au début du document.

Node:Utiliser les données, Next:[Utiliser les attributs](#), Previous:[Itérateur XML](#), Up:[Top](#)

Utiliser les données

On accède au contenu contenu d'un élément avec deux méthodes de balises, `getData` et `setData`:

```
text getData()           retourne le contenu.  
void setData(text)      change le contenu.
```

Ces méthodes sont associées au nom d'un document ou d'une instance, et concernent l'élément actuellement pointé.

```
nomdocument.getData()  
instance1.setData("quelque texte")
```

Node:Utiliser les attributs, Next:[XML et fonction](#), Previous:[Utiliser les données](#), Up:[Top](#)

Utiliser les attributs

On accède à un attribut comme à un élément de dictionnaire. L'attribut est reconnu par son nom en paramètre de la fonction `getAttribute()` ou `setAttribute()`.

```
xml doc  
  elem1 att = "valeur"  
/elem1  
/xml  
  
print doc.getAttribute("att1")  
doc.setAttribute("att1", "nouvelle valeur")
```

Node:XML et fonction, Next:[XML dynamique](#), Previous:[Utiliser les attributs](#), Up:[Top](#)

XML et fonction

Vous pouvez utiliser une instance de document XML comme argument d'une fonction, et en retourner par une fonction.

L'argument est une nouvelle instance et non pas un pointer sur l'instance originelle, et l'objet retourné est aussi une nouvelle instance.

Un document XML peut être défini à l'intérieur d'une fonction. Si le document doit être retourné par la fonction, c'est un pointeur sur le document qui sera retourné, pour augmenter la vitesse d'exécution.

```
xml dom  
/xml  
  
dom mafonction()  
  xml demo
```

```
        nomelement
            "contenu"
    /
/xml
return demo

dom x = mafonction()
print mafonction()
```

Node:XML dynamique, Next:[Méthodes XML](#), Previous:[XML et fonction](#), Up:[Top](#)

XML dynamique

Ce chapitre décrit les fonctions pour modifier la structure d'un document XML ou d'une instance.

`void clear()` Efface le contenu du document.

XML addChild(XML)

Ajoute un descendant à l'élément courant du document. Le nouvel élément devient l'élément courant.

```
xml demo
    doc /
/xml

xml element
    inner
        "quelque texte"
    /inner
/xml

demo.addChild(element)
```

xml addNext(xml)

Ajoute un élément après l'élément courant du document.

Le nouvel élément devient l'élément courant pour le document.

Node:Méthodes XML, Next:[Include](#), Previous:[XML dynamique](#), Up:[Top](#)

Méthodes XML

Méthodes

Définitions

<code>xml addChild(xml)</code>	Ajoute un descendant à l'élément courant.
<code>xml addNext(xml)</code>	Ajoute un successeur à l'élément courant.
<code>xml append(xml)</code>	Ajoute un autre document à la fin du document. Retourne le document augmenté.
<code>xml at()</code>	Retourne l'élément pointé, sous forme de document XML.
<code>xml at(text)</code>	Pointer sur un élément dans le niveau, son nom en paramètre.
<code>xml at(text, text)</code>	Pointer sur un élément trouvé par l'attribut et sa valeur, en paramètres.
<code>xml begin()</code>	Pointer le début du niveau. Concrètement, sur le child du parent de la balise courante.

<code>void clear()</code>	Efface le contenu du document.
<code>xml clone(xml)</code> paramètre.	Copie le contenu du document dans le document en paramètre.
<code>void display()</code>	Affiche le document.
<code>xml down()</code> courante.	Aller sur le premier sous-élément de la balise courante.
<code>boolean empty()</code>	Retourne true si le document ne contient aucun élément.
<code>xml end()</code>	Aller sur la dernière balise du niveau.
<code>boolean found()</code> false sinon.	Retourne true si dec ou inc ont atteint une balise, false sinon.
<code>text getData()</code>	Retourne le contenu de la balise courante.
<code>text getValue()</code>	Retourne la valeur de l'attribut en argument.
<code>boolean hasAttribute(text)</code>	Retourne true si l'attribut en argument existe dans la balise courant.
<code>xml inc()</code>	Pointe la balise suivante.
<code>int length()</code>	Retourne le nombre de balise dans le niveau.
<code>boolean load(text)</code> déclaré.	Charge le fichier nommé en argument dans le document déclaré.
<code>xml reset()</code>	Va au premier élément du document.
<code>boolean save(text)</code> Retourne true si ok.	Sauve le document dans un fichier nommé en argument.
<code>void setData(text)</code>	Modifie le contenu d'une balise.
<code>void setValue(text, text)</code>	Modifie la valeur d'un attribut, ou le crée.
<code>xml up()</code> balise courante.	Va au premier élément qui suit le conteneur de la balise courante.
<code>void xml([xml])</code> source.	Constructeur, avec en option un autre document comme source.

Node:Include, Next:[Define](#), Previous:[Méthodes XML](#), Up:[Top](#)

Include

La syntaxe pour inclure un fichier Scriptol externe est:

```
include "nomfichier.sol"
```

Les parenthèses sont optionnelles. Les guillemets simples ou doubles sont requis. Si vous voulez utiliser directement un fichier PHP, et que le compilateur ne le traite pas, voir ci-dessous.

Seuls les fichiers avec l'extension ".sol" seront traités par les compilateur, si elle est différente:

- solp (php) ignorera l'instruction.
- solc (c++) passera l'instruction au compilateur C++.

Si, au contraire, vous vouliez inclure un fichier dans le code PHP généré et pas dans le code C++, vous pouvez utiliser la fonction `require_once("nom de fichier")` qui sera ignorée par le compilateur en C++ solc, car l'insertion dynamique ne peut se faire en C++.

Node:Define, Next:[Utiliser une fonction comme variable](#), Previous:[Include](#), Up:[Top](#)

Define

Cette instruction permet de définir de nouveaux types que le parser puisse reconnaître, comme des primitives ou des classes.

Utiliser une fonction comme variable

Vous pouvez utiliser une fonction comme argument d'une autre fonction en la définissant en tant que type. Cela ne fonctionne qu'au niveau global et pas dans une classe. Une fois un type défini, on ne peut le redéfinir.

1) définir une fonction:

Exemple:

```
int compare(int a, int b)
  boolean b = (a < b)
  return b
```

2) définir un type, en utilisant cette fonction comme modèle:

```
define COMP = "compare"
```

3) définir une autre fonction générique qui utilisera cette fonction comme argument:

```
void mafonc(COMP a, int x, int y)
  print a(x,y)
  return
```

4) utiliser la fonction générique:

L'argument peut être la fonction originelle ou une autre fonction avec mêmes arguments et même types de retour.

```
mafonc("compare2", 10, 8)
```

Vous pouvez maintenant définir une autre fonction, "mul" par exemple, selon le même modèle, donc avec le même type de retour et d'arguments, que la fonction "add", et l'utiliser à la place.

```
mafonc("mul", x, y)
```

Utiliser un type externe

La syntaxe:

```
define NOUVEAUTYPE
define NOUVEAUTYPE as
```

crée un nouveau type que l'on peut utiliser en argument de fonctions externes. Voir les exemples GTK sur l'utilisation d'un type externe.

Le modificateur "as" fait que le nouveau type n'est pas considéré comme un pointeur. Donc l'utiliser comme une primitive, comme uint ou gint par exemple.

Utiliser les librairies standards

Librairies PHP

Scriptol peut utiliser les fonctions PHP. Aucun contrôle n'est effectué sur les arguments et types de retour des fonctions PHP.

Il faut juste configurer php.ini pour rendre les extensions visibles à l'interpréteur PHP.

Les fonctions PHP les plus usuelles sont incluses dans la librairie Scriptol.

Pour utiliser les fonctions de librairies PHP dans un programme exécutable, il faudra écrire une interface à inclure dans le projet. phpgd.h et phpgd.cpp en est un exemple.

Librairies C

Une bibliothèque de fonctions en C ou C++, peut être utilisée simplement par l'ajout du fichier ".lib" ou ".a" ou ".so" à la liste dans le fichier de configuration ".ini" ou ".cfg".

Un fichier à inclure doit être écrit pour rendre visible au compilateur les variables et les classes de l'extension ajoutée.

L'instruction "define" permet de créer un type pour donner une fonction en paramètre d'une autre fonction.

Il est possible de déclarer directement des variables C:

```
extern
  ~~extern char *message~~      pour le fichier d'en-tête.
  char *message                 pour la visibilité en scriptol.
/extern

~~char *message~~              création effective de la variable C.
```

Equivalences:

C	Scriptol
char *	cstring
void *	object
char **	cstring *
unsigned char	byte

- Un nom de fonction foncname en argument est écrit "foncname", voir define.
- Une fonction interne ne peut être utilisée en argument. Il faut l'encapsuler dans une fonction définie par l'utilisateur.
- Si une instance est déclarée sous la forme C++: NomClasse instance, il faut créer un pointeur plutôt (selon la forme C++: NomClasse *instance).

Fonctions usuelles

Ces fonctions sont communes à PHP, C, C++ et Scriptol. Si le nom PHP diffère, il est donné dans la liste.

text chr(integer)	Retourne le caractère pour une valeur ASCII. Ex: chr(32) retourne un espace blanc.
void die(text message)	Affiche un message en quittant le programme.
void exit()	Quitte le programme. Peut afficher un message.
number min(number, number)	Retourne le plus petit de deux scalaires.
number max(number, number)	Retourne le plus grand de deux scalaires.
int ord(text)	Retourne la valeur ASCII d'un caractère.
constant cstring plural(int x)	Retourne le pluriel "s" si le nombre x > 0.
array range(int x, int y)	Génère un tableau des entiers compris entre x et y.
cstring str(number)	Convertit un nombre en chaîne de caractères.
void swap(dyn, dyn)	Echange le contenu de deux variables.
text pad(text t, len l [, text c] [, int o])	Complète un texte avec des espaces ou la chaîne de caractères donnée. t: text à compléter. l: longueur à atteindre. c: texte à ajouter, des espaces par défaut. o: option STR_PAD_LEFT, STR_PAD_BOTH, par défaut à droite. (Voir: str_pad)

Fonctions de fichiers (voir aussi les méthodes du type File):

void exec(text)	Passe une commande au système d'exploitation.
boolean file_exists(text)	Teste si le fichier dont le nom est en argument existe.
number filesize(text)	Retourne la taille.
number filetime(text)	Retourne la taille (utiliser la fonction date pour afficher).
text filetype(text)	Retourne "dir" ou "file".
boolean rename(text, text)	Renomme un fichier. Retourne faux en cas d'échec.
void system(text commande)	Passe une commande au système d'exploitation.
boolean unlink(text)	Efface un fichier. Retourne true si effacé.

Fonctions de répertoire:

boolean chdir(text)	Change le répertoire courant. Retourne false si échec.
boolean mkdir(text)	Crée un sous-répertoire. Retourne true si créé.
boolean rmdir(text)	Efface un sous-répertoire. Retourne true si effacé.
text getcwd()	Retourne le chemin répertoire courant.

Fonctions mathématiques:

number abs(number)	Retourne la valeur absolue d'un nombre.
real acos(real)	
real asin(real)	
real atan(real)	
number ceil(number)	Retourne le nombre arrondi à l'entier supérieur.
real cos(real)	
real exp(real)	
number floor(number)	Retourne le nombre arrondi à l'entier inférieur.
number fmod(number, number)	Return le modulo de deux nombres.
real log(real)	
number pow(number, number)	Retourne la puissance n d'un nombre.
int rand()	Retourne un nombre aléatoire.
void randomize()	Démarré une séquence de nombres aléatoires.
real round(real)	Arrondi au plus proche, plancher ou plafond.
real sin(real)	

number sqrt(number) Retourne la racine d'un nombre.
real tan(real)

Fonctions de temps:

int time() Temps en millisecondes depuis le 1 Janvier 1970.
dict localtime() Temps et date courant lors de l'appel, dans un
dictionnaire, voir ci-dessous.

Clé du dict retourné par localtime:

tm_sec	Secondes après la minute	[0,61]
tm_min	Minutes après l'heure	[0,59]
tm_hour	Heures après minuit	[0,23]
tm_mday	Jour du mois	[1,31]
tm_mon	Mois à partir de Janvier	[0,11]
tm_year	Année depuis 1900	
tm_wday	Jour à partir de Dimanche	[0,6]
tm_yday	Jour depuis le 1 Janvier	[0,365]
tm_isdst	Indicateur d'heure d'hiver	

Node:XML intégré, Next:[Appendice I](#), Previous:[Fonctions usuelles](#), Up:[Top](#)

XML intégré (Scriptol C++ only)

Ce chapitre sera remplacé par les chapitres "XML statique" et "XML dynamique".

Ce qui est décrit ici est fonctionnel dans le compilateur de Scriptol en C++, et restera compatible avec le nouveau mode d'utilisation de XML, plus simple, qui est actuellement implémenté dans l'interpréteur.

Dom

Pour utiliser le document XML, une instance de dom doit être créée. L'arborescence du dom est ensuite remplie par la méthode build() :

```
dom mondom
```

```
mondom.build()
```

La liste complète des méthodes de dom est donnée sur le tutoriel sur CD. Les principales sont décrites dans ce chapitre.

Pour afficher le document, utiliser la méthode display().

Pour le sauver, la méthode save() avec le nom de fichier en argument. Dans les deux cas, l'argument dom.LIGHT permet la présentation allégée. Vous pouvez assigner un document dans un dom à un autre dom avec la méthode assign.

Chemin

Une fois l'arborescence créée, vous pouvez accéder aux données et aux attributs des éléments avec les méthodes de dom.

Mais Scriptol a une syntaxe spéciale pour pointer sur un élément dans le document. Les éléments et éléments imbriqués sont traités comme des objets et des sous-objets:

La syntaxe est:

```
instance [.nomelement]* "[" nom-attribut : valeur]" ? [.nom-méthode(arguments)]
```

- le nom de l'instance de dom,
- un point suivi par le nom de l'élément,

- ceci pour chaque sous-élément,
- un couple attribut : valeur entre crochets, si nous voulons sélectionner un élément parmi plusieurs de même nom,
- un point suivi par une méthode et ses arguments.

Si l'on veut juste pointer sur un élément, on utilise la méthode `at()`.

Mais l'on peut ajouter toute autre méthode de `dom` au chemin d'un élément. Voir `demoxml.sol` pour exemple.

Lecture et écriture

On peut lire et changer la valeur d'un attribut, de même pour les données.

- `getValue(nom-attribut)` retourne la valeur de l'attribut en argument.
- `setValue(nom-attribut, valeur)` assigne une valeur à un attribut.
- `getData()` retourne les données.
- `setData(texte)` change ou assigne le texte donné en paramètre.

Les données peuvent être aussi assignées directement:

```
mondom.unelement.souselement.setData("quelque texte") ou
mondom.unelement.souselement = "quelque texte"
```

Itérateur

La classe `dom class` peut aussi utiliser le document comme une pile XML.

Plusieurs méthodes forment l'itérateur.

- `begin()` pointe sur le premier élément du document.
- `next()` déplace sur l'élément suivant (dans le même élément conteneur).
- `up()` déplace sur le successeur du conteneur.
- `down()` pointe sur le sous-élément de l'élément courant.
- `found()` retourne vrai si le successeur ou sous-élément existe, faux autrement.

Une fois un élément pointé, on accède aux données avec les méthodes suivantes.

Insérer et supprimer des éléments

Vous pouvez insérer des éléments soit comme sous-élément ou comme successeur d'un élément pointé, ou supprimer un élément pointé.

Syntaxe:

```
nominstance.chemin.addNext(xnode)
nominstance.chemin.addChild(xnode)
nominstance.chemin.remove()
```

Pour insérer un élément, vous devez le déclarer dans un autre document XML. Vous pouvez déplacer un élément en l'insérant à une nouvelle position, puis en supprimant l'occurrence originelle.

HTML

Une page HTML a ce format:

```
xml
  html
    ...content...
  /html
/xml
```

Appendice I: Utiliser l'API Java API avec Scriptol

Scriptol implémente la déclaration d'objets Java, grâce au modifieur "java", et pourvu que l'extension Java soit activée (voir la fiche: install).

De la même façon que l'on inclut les fichiers dont on veut utiliser le contenu, il faut importer les fichiers d'une classe Java pour l'utiliser.

Une fois la classe importée, on peut déclarer des instances et appeler ses méthodes comme pour une classe Scriptol.

Le fichier source java doit être compilé par le compilateur java en .class avant d'être importé.

La syntaxe est:

```
import java cheminclasse-nomclasse
ou
import
... déclarations de classes ...
/import
```

Si le mot "java" est au début du chemin de la classe, le compilateur l'identifie et le modifieur peut être omis:

Exemples:

```
import java java.awt.Dialog      déclaration valide
import java.awt.Dialog          est aussi valide
import java MaClasse            déclaration valide
import MaClasse                 n'est pas valide pour une classe Java.
```

Déclarer une instance d'une classe Java est comme déclarer une instance d'une classe Scriptol.

Exemple:

```
java.awt.Dialog maDialogue      ... ou simplement
Dialog monDialogue
MaClass monInstance
```

Si "Dialog" n'est déclaré comme classe Scriptol vous pouvez omettre le chemin.

Une fois l'instance déclarée, vous l'utilisez comme tout autre objet.

```
monDialogue.setVisible(true)
monInstance.disp()
print monInstance.x
```

Si vous écrivez votre propre classe Java, elle doit être dans un fichier séparé portant le nom de la classe, avec l'extension .java. Elle doit être compilée par javac.

Appendice II: Gestion des exceptions

Le traitement des exceptions n'est supporté que par les langages cibles C++ et PHP 5.

Syntaxe:

```
extern
  class exception
    string what()
  /class
/extern

try
  ... quelques instructions ...
catch(exception e)
  print e.what()
/try
```

Node:Appendice III, Next:[Appendice IV](#), Previous:[Appendice II](#), Up:[Top](#)

Appendice III: Langues étrangères

Le préprocesseur permet d'écrire des programmes scriptol avec des mot-clés dans toute langue.

Pour l'utiliser, il faut:

- 1) une liste de mot-clés.
- 2) l'option -t en ligne de commande (ne marche pas avec l'option -w).
- 3) une entrée dans le fichier solc.ini ou solp.ini (ou .cfg) de la forme:

Keywords=extension

L'extension est un code de deux lettres selon la langue utilisée, exemple:

Keywords=fr ...pour la langue française.

La liste originale des mot-clés est dans le fichier "keywords.en". S'il n'y a pas de liste dans la langue que vous voulez utiliser il faut la créer avec chaque mot étranger suivi par l'équivalent anglais (prendre keywords.fr pour exemple).

Si le fichier ".ini" ou ".cfg" propre à un source n'a pas d'entrée Keyword, l'option -t sera ignorée pour ce fichier.

Node:Appendice IV, Next:[Index](#), Previous:[Appendice III](#), Up:[Top](#)

Appendice IV: Syntaxe obsolète

- les symboles <- et ->.
 - le mot-clé string est remplacé par cstring.
 - les déclarations externes char * sont remplacées par cstring.
 - l'opérateur match et sa définition sont obsolètes.
 - les méthodes sur les nombres sont inutiles et supprimées.
 - utiliser () pour un tableau vide est dépassé. Utiliser plutôt {} ou array().
 - le préfixe pour une chaîne PHP est dépassé et inutile.
-

Index

- Fonctions: [Fonctions](#)
- HTML: [Page HTML](#)
- XML: [XML intégré](#), [Utiliser les attributs](#), [Définir un document XML](#), [XML light](#)
- afficher: [Print et echo](#)
- alias: [Alias](#)
- aléatoire: [Assignement direct](#)
- and: [Expression](#), [Opérateurs](#)
- application: [Projet Scriptol](#)
- argument: [Alias](#)
- arithmétique: [Opérateurs](#), [Expression](#)
- array: [Variables ou primitives](#), [Expression](#), [Tableau](#), [Tableaux typés](#), [Méthodes de array et dict](#), [Contenu d'un tableau](#), [Créer un tableau](#), [Parcourir un tableau à deux dimensions](#), [Variable dynamique](#)
- assignement: [Assignement simple](#), [Assignement conditionnel](#), [Assignement](#), [Assignement direct](#), [Assignement et conversion](#)
- associative: [Séquence et liste](#), [Dictionnaire](#)
- at: [Mode silencieux](#)
- attribut: [Méthodes et attributs statiques](#), [Class](#), [Définir une classe](#)
- augmenté: [Assignement augmenté](#)
- binaire: [Opérateurs de tableau](#), [Opérateurs](#), [Expression](#), [Compiler un programme Scriptol en binaire](#)
- bloc: [Scopes](#)
- boolean: [Variables ou primitives](#)
- boucle: [Structures de contrôle](#)
- break: [Break et continue](#), [Do case](#)
- c: [Librairies C](#)
- c++: [Utiliser les librairies standards](#), [Compiler un programme Scriptol en binaire](#), [Insérer du code cible](#)
- caractéristiques: [Caractéristiques du langage](#)
- case: [Do case](#)
- chargement: [Chargement et sauvegarde](#)
- class: [Class](#), [Scopes](#)
- classe: [Classe externe](#)
- clavier: [Input](#)
- commande: [Interpréter un programme scriptol](#), [Compiler un programme Scriptol en PHP](#)
- comment: [Mon premier programme](#)
- commentaire: [Commentaire](#)
- compatibilité: [Contenu d'un tableau](#), [Indexer un tableau](#), [Appendice IV](#), [Limitations et compatibilité](#)
- compilateur: [Compiler un programme Scriptol en binaire](#)
- compiler: [Compiler un programme Scriptol en binaire](#), [Compiler un programme Scriptol en PHP](#)
- composite: [If composite](#)
- composé: [Assignement augmenté](#)
- conditionnel: [Assignement conditionnel](#)
- constant: [Enum](#), [Variables et constantes](#), [Constante](#)

- constante: [Variables et constantes](#)
- constructeur: [Constructeur et littéral](#), [Constructeur et instance](#)
- continue: [While let](#), [Break et continue](#)
- contrôle: [Structures de contrôle](#)
- conversion: [Assignement et conversion](#)
- dec: [Itérateur](#)
- define: [Define](#)
- description: [Généralités](#)
- dict: [Dictionnaire](#), [Variables ou primitives](#), [Méthodes de array et dict](#)
- dimension: [Tableau à plusieurs dimensions](#)
- dimensions: [Parcourir un tableau à deux dimensions](#)
- dir: [Dir](#)
- do: [Do case](#), [Syntaxe étendue de do](#)
- dom: [XML intégré](#)
- dyn: [Variable dynamique](#), [Tableau typé et dyn](#)
- dynamique: [Tableau typé et dyn](#)
- déclaration: [Déclaration](#)
- défaut: [Valeurs par défaut](#)
- définition: [Définir une classe](#)
- démarrer: [Mon premier programme](#)
- echo: [Print et echo](#)
- entier: [Tableau d'entiers](#)
- entrée: [Input](#)
- enum: [Enum](#)
- erreur: [La structure de contrôle error](#)
- error: [La structure de contrôle error](#)
- et: [Expression](#)
- exception: [Appendice II](#)
- expression: [Expression](#)
- extern: [Extern](#)
- externe: [Extern](#), [Types externes](#), [Utiliser les bibliothèques standards](#)
- extraction: [Intervalles dans une liste](#)
- fichier: [File](#)
- file: [File](#), [Variables ou primitives](#)
- flot: [Structures de contrôle](#)
- fonction: [Utiliser un type externe](#), [Utiliser une fonction comme variable](#), [Fonctions usuelles](#), [Fonction](#)
- for: [For in](#), [Opérateurs de tableau](#)
- français: [Appendice III](#)
- global: [Scopes](#)
- guillemet: [Variable dans une chaîne](#), [Guillemets et échappement](#)
- héritage: [Héritage](#)
- identifieur: [Identifieurs et mot-clés](#)
- if: [If](#)
- imbriqué: [Scopes](#)
- in: [Opérateurs de tableau](#)
- inc: [Itérateur](#)
- include: [Include](#)
- index: [Indexer un tableau](#)
- indexation: [Indexer un dict](#), [Indexation](#)

- indice: [Indexer un tableau](#), [Indexation](#), [Intervalle](#)
- input: [Input](#)
- insertion: [Intervalles dans une liste](#)
- instance: [Constructeur et instance](#), [Tableau d'objets](#), [Class](#)
- instruction: [Instruction](#)
- integer: [Tableau d'entiers](#), [Variables ou primitives](#)
- interpréter: [Interpréter un programme scriptol](#)
- intersection: [Opérateurs](#), [Expression](#), [Opérateurs de tableau](#)
- intervalle: [Intervalle dans un dict](#), [Intervalle](#), [Intervalle dans un tableau](#), [Intervalles dans une liste](#)
- intégré: [Fonctions usuelles](#)
- intérieur: [Scopes](#)
- iterator: [Itérateur](#)
- itérateur: [Itérateur XML](#)
- itération: [Structures de contrôle](#)
- java: [Appendice I](#)
- langage: [Caractéristiques du langage](#)
- let: [While let](#)
- librairie: [Librairies PHP](#), [Librairies C](#), [Utiliser les librairies standards](#)
- limitation: [Limitations et compatibilité](#)
- lire: [Read](#)
- liste: [Séquence et liste](#), [Expression](#)
- littéral: [Littéraux](#), [Créer un dictionnaire](#), [Constructeur et littéral](#)
- local: [Scopes](#)
- logique: [Opérateurs](#), [Expression](#)
- léger: [XML intégré](#)
- main: [Valeurs par défaut](#), [Main](#)
- membre: [Définir une classe](#)
- mot-clé: [Identifieurs et mot-clés](#)
- multi-dimensionnel: [Tableau à plusieurs dimensions](#)
- multiple: [Assignements multiples](#)
- méthode: [Méthodes XML](#), [Méthodes et attributs statiques](#), [Méthodes de text](#), [Class](#), [Définir une classe](#)
- natif: [Compiler un programme Scriptol en binaire](#), [Insérer du code cible](#)
- naturel: [Tableau de number](#)
- nil: [Nil et null](#)
- nombre: [Tableau de number](#)
- not: [Expression](#), [Opérateurs](#)
- null: [Nil et null](#)
- number: [Variables ou primitives](#), [Tableaux typés](#), [Tableau de number](#)
- objet: [Définir une classe](#), [Tableau d'objets](#), [Class](#)
- obsolète: [Appendice IV](#)
- opérateur: [Opérateurs de tableau](#), [Symboles](#), [Opérateurs](#)
- or: [Opérateurs](#), [Expression](#)
- ou: [Expression](#)
- overloading: [Surcharge](#)
- php: [Utiliser les librairies standards](#), [Page HTML](#), [Compiler un programme Scriptol en PHP](#), [Variable dans une chaîne](#), [Indexer un tableau](#), [Librairies PHP](#), [Insérer du code cible](#), [Contenu d'un tableau](#)
- polymorphisme: [Surcharge](#)
- premier: [Mon premier programme](#)

- primitive: [Variables ou primitives](#)
- print: [Print et echo](#)
- projet: [Projet Scriptol](#)
- précédence: [Opérateurs](#), [Précédence](#), [Expression](#)
- range: [Intervalles dans une liste](#), [Intervalle](#)
- read: [Read](#)
- readline: [Read](#)
- real: [Variables ou primitives](#), [Tableau de number](#)
- relationnel: [Expression](#)
- répertoire: [Dir](#)
- sauvegarder: [Chargement et sauvegarde](#)
- scalaire: [Variables ou primitives](#)
- scan: [Scan by](#), [Opérateurs de tableau](#)
- scope: [Scopes](#), [Scope et fonction](#)
- scriptol: [Généralités](#)
- silencieux: [Mode silencieux](#)
- slicing: [Intervalles dans une liste](#)
- source: [Fichier source Scriptol](#), [Interpréter un programme scriptol](#)
- splicing: [Intervalles dans une liste](#)
- stack: [Utiliser un tableau comme pile](#)
- standard: [Librairies C](#)
- static: [Méthodes et attributs statiques](#)
- structure: [Structures de contrôle](#), [Scopes](#)
- subscripting: [Indexation](#), [Intervalle](#)
- sujet: [Au sujet de ce manuel](#)
- suppression: [Intervalles dans une liste](#)
- surcharge: [Surcharge](#)
- symbole: [Symboles](#)
- tableau: [Parcourir un tableau à deux dimensions](#), [Tableau](#)
- text: [Tableaux typés](#), [Tableau de textes](#), [Variables ou primitives](#), [Text](#), [Expression](#)
- texte: [Tableau de textes](#), [Text](#)
- type: [Tableaux typés](#), [Class](#), [Define](#), [Types externes](#), [Variables ou primitives](#)
- typed: [Tableaux typés](#)
- union: [Opérateurs](#), [Opérateurs de tableau](#), [Expression](#)
- until: [Do until](#)
- using: [Utiliser les librairies standards](#)
- variable: [Utiliser un type externe](#), [Scopes](#), [Utiliser une fonction comme variable](#), [Variables ou primitives](#), [Variable dans une chaîne](#), [Variables et constantes](#)
- visibilité: [Scope et fonction](#), [Scopes](#)
- web: [Page HTML](#)
- while: [While](#), [Syntaxe étendue de do](#)
- wrapper: [Define](#)
- write: [Write](#)
- xml: [Chargement et sauvegarde](#), [Itérateur XML](#), [Utiliser les données](#), [XML et fonction](#), [XML dynamique](#), [Méthodes XML](#), [Assignement XML](#), [XML statique](#)
- échappement: [Guillemets et échappement](#)
- écrire: [Write](#)
- éditeur: [Mon premier programme](#)
- étranger: [Appendice III](#)

