MOGWAI basics

Stéphane SIBUE - 2025

MOGWAI

Table des matières

MOGWAI

INTRODUCTION

Je développe depuis très longtemps et j'ai eu l'occasion d'utiliser beaucoup de technologies différentes et des langages très variés, mais je crois que le langage qui m'a le plus « parlé » c'est le RPL.

RPL c'est l'acronyme de Reverse Polish Lisp, et c'est le nom d'un langage créé par HP pour ses calculatrices scientifiques et financières.

Calculatrice HP 48SX programmable en RPL

RPL ressemble beaucoup à FORTH. Comme lui il utilise une pile pour prendre les paramètres et y déposer les résultats, et comme lui il utilise la Notation Polonaise Inverse (Reverse Polish Notation soit RPN, ne pas confondre avec le langage RPL bien sûr). Ainsi en RPN on n'écrit pas 2+2 pour effectuer une addition, mais 2 2 +, c'est un peu déroutant au début, mais avec cette notation plus besoin de parenthèses ni de variables locales (en théorie).

Bien sûr, la saisie des programmes RPL se faisait directement sur la machine, et l'ergonomie n'était pas au top, mais HP avait mis en place toute une batterie d'astuces et de fonctions pour rendre l'exercice vivable. Au premier abord, le RPL n'a pas une syntaxe très simple, mais en creusant le sujet on se rend vite compte de la puissance qu'il peut dégager.

```
«
  →STR STR→ DUP
  → N
  «
    "0"
    WHILE
      N DUP 2 / IP 'N' STO
    REPEAT
      DUP +
    END
    1 ROT SUB +
  »
»
```

Une opportunité à saisir

En fait, l'idée de lancer le développement de MOGWAI est venue le jour où nous avons eu besoin, au boulot, de pouvoir simuler un périphérique Bluetooth Low Energy. Quand on développe une application mobile (c'est mon métier) qui utilise la communication Bluetooth Low Energy pour communiquer avec un appareil donné, l'appareil en question n'existe pas encore car Il doit d'abord être conçu physiquement et son logiciel interne devra ensuite être écrit, testé et validé.

Toute cette procédure prend du temps et généralement, pour ne pas trop en perdre, nous commençons le développement de l'application mobile bien avant que la carte électronique soit en mesure d'échanger la moindre information. Garder « pour la fin » la partie communication BLE n'est pas une bonne idée. En effet, pour une intégration idéale et pour le plus de personnes possibles soit en mesure d'utiliser les yeux fermés l'application (communication incluse) il faut intégrer la dimension BLE dès le début.

Alors nous nous sommes développés un outil qui permet de simuler le fonctionnement d'un appareil communiquant en BLE avant même qu'il existe. Cela permet de valider bien en amont les échanges à mettre en place et cela permet aussi de se rendre compte très tôt de toutes les petites choses qui n'avaient pas été correctement prévues. C'est donc un outil très important pour obtenir une application mobile robuste en termes de communication BLE. De plus, cela permet à la partie électronique et embarquée de valider très tôt des choix cruciaux en matière de communication via le Bluetooth Low Energy.

Avec ce type de moteur, les fonctions « profondes » du simulateur restent très génériques en effectuant toutes les opérations nécessaires sous la directive d'un code qui est modifiable à volonté, en temps réel, sans recompilation, car c'est le code du moteur d'exécution qui va s'occuper de toute la partie « logique » de la simulation. Il suffira de stocker pour chaque périphérique un jeu de scripts adapté à son mode de fonctionnement et adapté aux tests à réaliser. La souplesse obtenue était énorme !

Le simulateur doit pouvoir exécuter des instructions très variées. Il doit être en mesure de générer la structure du périphérique BLE à simuler, et aussi effectuer des tâches qui vont le faire réagir comme si c'était le vrai périphérique. Pour cela il faut idéalement pouvoir « programmer » le simulateur. Et c'est pour cet usage de base que MOGWAI a été développé. C'est un moteur d'exécution qu'il est possible d'inclure dans une application qu'il faut « motoriser ».

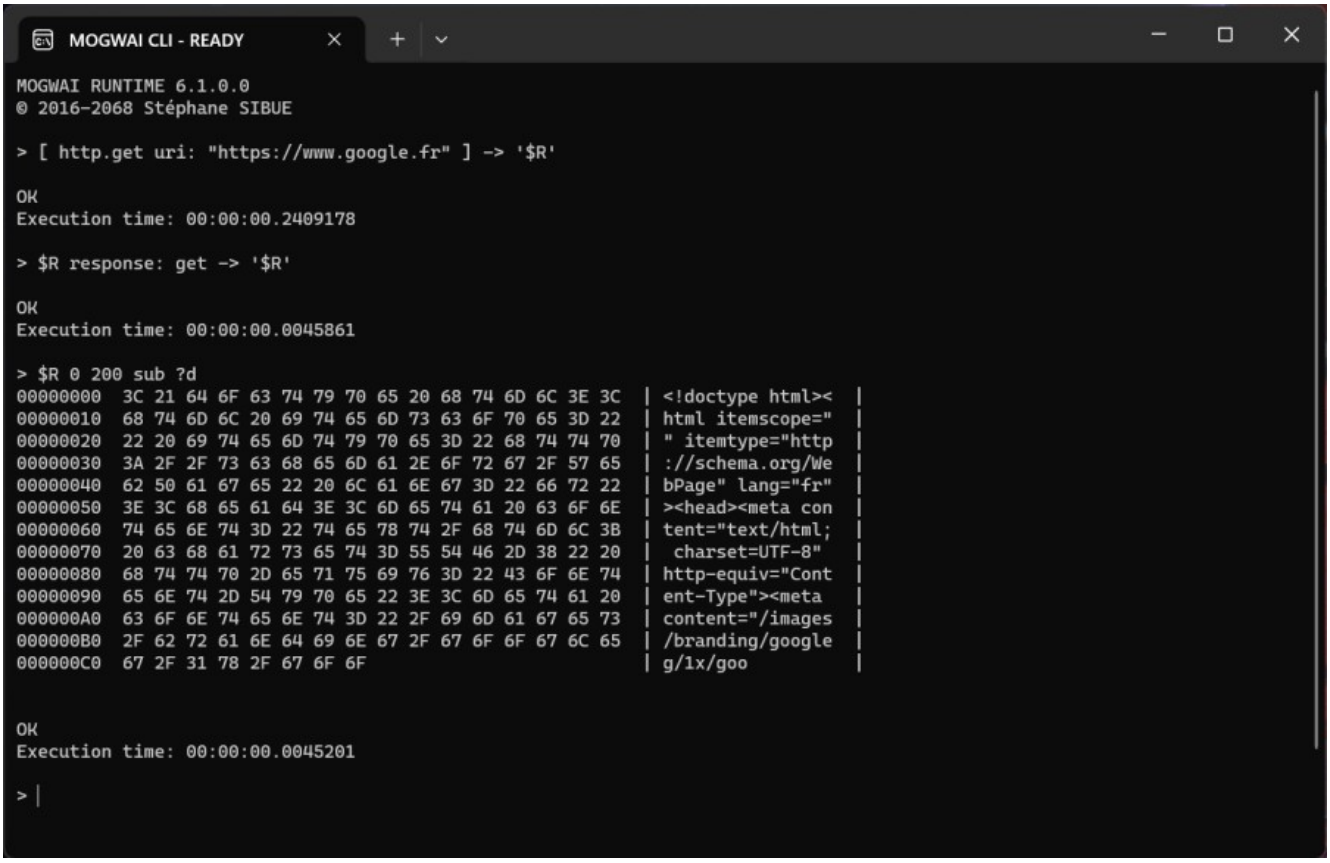
Le simulateur BLE était le projet idéal pour lancer le développement de MOGWAI.

Une lente maturation

La 1ère version de MOGWAI a été développée en.NET Standard avec le langage C#. La bibliothèque MOGWAI a été incluse dans le simulateur qui était développé en UWP. Comme le simulateur devait prendre le rôle d'un périphérique BLE, il fallait une machine dotée d'une puce BLE capable de prendre en charge ce rôle (généralement les puces BLE des PC de bureau ne savent prendre en charge que le rôle de Central BLE). Les Raspberry PI 3 sont dotés d'une puce BLE capable de prendre les 2 rôles. En installant un Windows 10 IOT sur un Raspberry PI 3 nous avons pu faire tourner sans problème la 1ère version du simulateur, motorisé par la 1ère version de MOGWAI. Cet outil nous a fait gagner beaucoup de temps à l'époque.

Au fur et à mesure des besoins du simulateur BLE, le moteur MOGWAI a été étendu, amélioré, et beaucoup de nouvelles fonctionnalités ont été ajoutées. Aujourd'hui MOGWAI sait gérer les liaisons séries, les requêtes HTTP, les bases de données SQLite et possède plus de 200 primitives.

J'en suis à la version 6, toujours développées en C# pour .NET. Ceci permet de l'utiliser sous Windows, mais aussi sous Linux et Mac OSX avec des architectures X86, X64 et ARM. Par exemple MOGWAI fonctionne nativement sur un Raspberry PI 3 sous Raspbian (Linux ARM).



MOGWAI CLI pour utiliser le langage en mode interactif

Pour « jouer » avec MOGWAI j'ai développé une application console en mode interactif qui permet d'utiliser toutes les fonctionnalités du langage.

Il est tout à fait possible de saisir les programmes MOGWAI avec un simple bloc-notes, mais il est tout de même plus agréable d'avoir des outils de développement adaptés. MOGWAI Studio est un IDE de développement dédié à MOGWAI. Il est en cours de développement mais fonctionne déjà dans les grandes lignes (colorisation syntaxique, fonctions de base de débogage, mode interactif intégré par exemple).

PARTIR DU BON PIED

Il y a un reflex à adopter avec MOGWAI, c'est de placer en 1ère instruction de vos programmes la fonction `mogwai.reset`.

Elle assure d'avoir un moteur d'exécution absolument vierge, aucune variable, aucun timer, aucune tâche, rien de rien.

Par exemple, l'application MOGWAI CLI qui permet de "jouer" avec MOGWAI ne remet jamais à zéro le contexte d'exécution, ce qui signifie que tout ce que vous créez au fur et à mesure des lignes que vous tapez est gardé, ce qui permet d'enchainer des commandes pour effectuer pas à pas les opérations à tester.

Donc n'oubliez pas, pour tout remettre à plat, utilisez la fonction `mogwai.reset`.

AFFICHAGE DE VALEURS

Il existe principalement 2 fonctions pour afficher des valeurs à l'écran.

`println` affiche l'objet qui se trouve en 1ère position sur la pile et effectue un retour à la ligne automatiquement. Pour gagner en concision il est possible d'utiliser `?` à la place.

`print` effectue la même opération sans retour à la ligne automatique. Cette fonction peut être remplacée par `??`.

```
# On affiche la valeur 15 et la chaîne "HELLO !" sur 2 lignes
15 ?
"HELLO !" ?

# On affiche le message "NOUS SOMMES EN 2025" en 2 parties, une chaîne et un nombre.
"NOUS SOMMES EN " ??
2025 ?
```

Pour effacer l'écran, il faut utiliser la fonction `cls`.

SAISIE A L'ECRAN

Pour saisir des données à l'écran il existe 2 fonctions, `input` et `prompt`.

La plus simple est `input` qui se met en attente de la saisie au clavier se terminant par un retour chariot (touche ENTER). Les informations saisies sont placées sur la pile sous la forme d'une chaîne de caractères.

```
# On passe en saisie et on stocke le résultat dans la variable $X

input -> '$X'
```

La fonction `prompt` fonctionne exactement comme `input` mais elle permet d'afficher un message d'invite en plus. Ce message est placé sur la pile avant d'appeler la fonction `prompt`.

```
# On demande la saisie du nom
# Et on stocke l'information dans la variable '$NOM"

"Quel est votre nom ? " prompt -> '$NOM'
```

VARIABLES

Les variables sont définies par un nom. Si le nom commence par le symbole `$`, elle sera globale, dans tous les autres cas, elle sera locale.

Par défaut, une variable n'a pas besoin d'être déclarée pour être utilisée. La 1ère affectation la crée si elle n'existe pas déjà.

Par défaut une variable n'a pas de type prédéfini, elle prend le type de la dernière valeur qui lui a été affectée mais il est possible de verrouiller le type d'une variable si on effectue une déclaration préalable à son utilisation. Elle prend alors le type déclaré et une erreur est levée si on essaye de lui affecter une valeur d'un autre type.

Les variables typées sont déclarées avec la fonction `def`.

```
# On affecte la valeur numérique 50 à la variable locale 'A'
50 -> 'A'

# On affecte ensuite une chaîne de caractères à cette variable
"Bonjour !" -> 'A'

# On affecte la valeur numérique 500 à la variable globale '$R'
500 -> '$R'

# On déclare la variable globale '$Z' avec le type .number
# Qui lui permettra de stocker des nombres
.number '$Z' def

# On ne peut maintenant stocker dans '$Z' que des nombres
1500 -> '$Z'

# Sinon une erreur est levée
"Bonjour !" -> '$Z'

# Si on veut déclarer une variable qui accepte de stocker n'importe quel type
# De valeurs, il faut utiliser le type .any
.any '$X' def
1500 -> '$X'
"Bonjour !" -> '$X'
```

Il est possible de rendre obligatoire la déclaration préalable des variables avant de les utiliser. Il suffit pour cela d'utiliser la fonction `explicit` avec true ou false pour activer ou désactiver cette obligation.

```
# On active la déclaration obligatoire des variables avant de les utiliser
true explicit
```

Lorsqu'une variable n'a plus besoin d'exister, il est possible de la supprimer explicitement en utilisant la fonction purge. Une variable locale sera quoi qu'il arrive supprimée automatiquement quand le code sort de son scope. Si on essaye de supprimer une variable qui n'existe pas, une erreur est levée.

```
# On supprime la variable locale A
'A' purge
```

Pour placer sur la pile la valeur d'une variable il suffit d'invoquer son nom sans les apostrophes.

```
# On affecte 'A' et 'B' avec des nombres.

20 -> 'A'

30 -> 'B'


# On effectue la somme des 2 variables et on stocke le résultat dans la variable 'C'

A B + -> 'C'
```

Avec la fonction rcl, il est possible de placer sur la pile la valeur d'une variable en utilisant son nom.

```
# On récupère la valeur d'une variable via son nom (avec les apostrophes).
100 -> 'A'
'A' rcl

# 100 est posé sur la pile.
```

Pour stocker dans une variable numérique le résultat d'une opération mathématique sur elle-même (genre on ajoute 1 à la valeur de la variable X), il existe 4 fonctions d'affectation supplémentaires :

->+	Ajoute un nombre à une variable. 100 -> 'A' 10 ->+ 'A' # Maintenant A vaut 110.
->-	Soustrait un nombre à une variable. 100 -> 'A' 10 ->- 'A' # Maintenant A vaut 90.
->*	Multiplie un nombre et une variable. 100 -> 'A' 10 ->* 'A' # Maintenant A vaut 1000.
->/	Divise un nombre et une variable. 100 -> 'A' 10 ->/ 'A' # Maintenant A vaut 10.

Si la variable n'existe pas elle est créée avec la valeur 0 par défaut, l'opération sera ensuite effectuée à partir de cette valeur.

Si la variable n'est pas numérique, elle est initialisée comme si elle n'existait pas avant.

Si la variable n'est pas de type numérique et a été déclarée (type verrouillé), une erreur est levée.

Pour gagner du temps il existe aussi 2 fonctions pour incrémenter et décrémenter une variable numérique.

`++`	Incrémente une variable.
	
	`100 -> 'A'`
	`'A' ++`
	`# Maintenant A vaut 101.`

`--`	Décrémente une variable.
	
	`100 -> 'A'`
	`'A' --`
	`# Maintenant A vaut 99.`

La fonction `-->` permet d'appliquer un traitement à la valeur d'une variable et de stocker le résultat dans cette même variable :

```
# $A contient la valeur 100
# On veut calculer la racine carrée multipliée par 5 de $A et stocker le résultat dans $A

# Méthode classique

100 -> '$A'
$A sqrt 5 * -> '$A'

# Ecriture plus rapide -->

100 -> '$A'
{sqrt 5 *} --> '$A'
```

La fonction `vars` retourne la liste de toutes les variables globales utilisées :

```
# On crée 3 variables globales $A, $B et $C

50 -> '$A'
100 -> '$B'
$A $B + -> '$C'

# On liste les variables globales utilisées

vars

# Pose sur la pile la liste ('$A' '$B' '$C')
```

La fonction `lvars` retourne la liste de toutes les variables locales utilisées :

```
# On crée 3 variables locales A, B et C

50 -> 'A'
100 -> 'B'
A B + -> 'C'

# On liste les variables locales utilisées

lvars

# Pose sur la pile la liste ('A' 'B' 'C')
```

Il est possible de vérifier l'existence d'une variable avec la fonction `exists`. Cette fonction retourne `true` si le nom de la variable passé en paramètre existe (variable locale ou globale).

```
# On crée 1 variable locale A

50 -> 'A'

'A' exists

# Pose sur la pile true
```

CONSTANTES

Il est possible de déclarer des constantes. Ce sont des variables dont le contenu ne peut plus être modifié après leur 1ère affectation.

Les constantes sont déclarées avec la fonction `const`.

```
# On définit une constante numérique
100 '$BUFFER_SIZE' const

# On définit une constante de type chaine de caractères
"XYZ45FGHH" '$USER_PASSWORD' const

# On crée une nouvelle chaine de caractères à partir de $USER_PASSWORD
# Que l'on stocke dans la variable 'pwd'
$USER_PASSWORD ".X1" + -> 'pwd'
```

LES TYPES

MOGWAI manipule des objets ayant des types différents.

Chaque type possède un nom qui commence par un point. Par exemple le type correspondant à la chaîne de caractères a comme nom `.string`.

La fonction `->type` permet de récupérer le type de l'objet qui se trouve sur la pile.

```
# Le type d'un nombre est .number
1567 ->type ?

# On peut tester le type d'une variable et prendre des décisions en conséquence
234 -> 'A'
if (A ->type .number ==) then {"A est un nombre" ?} else {"A n'est pas un nombre" ?}
```

Les principaux types manipulés par MOGWAI sont les suivants :

Nom	Type	Exemple
.number	Nombre (réel double précision)	154 ou -56.34
.string	Chaîne de caractères	"Bonjour le monde"
.boolean	Valeur booléenne	true / false
.list	Liste d'objets	(5 "X1" 12.78)
.block	Bloc de code	{2 2 + ?}
.program	Programme (fonction)	«2 2 + ?»
.name	Nom symbolique	'A'
.key	Clé utilisée dans un record	latitude:
.data	Tableau de bytes	DATA:FF3456ED23
.binary	Nombre binaire	BIN:110011110011
.record	Enregistrement (dictionnaire)	[x: 50 y: 200]
.null	Valeur nulle	null -> 'A'
.any	N'importe quel type (variant)	

MOGWAI

MOGWAI propose 9 fonctions qui permettent de tester plus facilement les types les plus utilisés :

Fonction	Test	Résultat
isnull	154 isnull	false
isRecord	[x: 50 y: 20] isRecord	true
isList	(1 2 3) isList	true
isName	'A' isName	true
isString	"Bonjour le monde" isString	true
isKey	X: isKey	true
isData	DATA:50FF4E isData	true
isNumber	564 isNumber	true
isBoolean	false isBoolean	true

LA PILE

MOGWAI est un langage qui utilise une pile LIFO pour fournir les paramètres aux fonctions et récupérer les résultats. On peut poser sur la pile n'importe quel objet manipulé par MOGWAI (voir le chapitre LES TYPES).

Par exemple quand vous écrivez 2 8 + pour effectuer une addition, MOGWAI va effectuer une série d'opérations lors de l'exécution :

1. Poser 2 sur la pile (2 se trouve en 1ère position)
2. Poser 8 sur la pile (8 se trouve en 1ère position, et 2 en seconde position)
3. Exécuter la fonction + qui va prendre les 2 valeurs en haut de la pile, les ajouter et poser le résultat sur la pile.

Au final sur la pile, 2 et 8 ont disparu (on dit qu'ils ont été consommés par la fonction +), remplacés par le résultat de leur somme (soit la valeur 10).

Affectation d'une variable

Quand vous assignez une valeur à une variable, c'est exactement le même processus qui se passe.

Par exemple, quand vous écrivez 500 -> 'A' vous utilisez une version édulcorée pour l'affectation (c'est plus pratique à écrire comme ça), mais en réalisé le parseur de MOGWAI transforme ces instructions en

```
500 'A' sto
```

La fonction sto est la fonction d'affectation qui prend 2 paramètres sur la pile, la valeur et le nom de la variable.

1. 500 est posé sur la pile
2. 'A' est posé sur la pile
3. La fonction sto prend la valeur et le nom et affecte à la variable correspondante la valeur fournie.

MOGWAI vous évite d'écrire certaines fonctions 100% en RPN car ce n'est pas une gymnastique simple.

Voici la liste des fonctions d'affectation ayant une version édulcorée :

Fonction édulcorée	Fonction 100% RPN	Usage édulcoré	Usage 100% RPN
->	sto	10 -> 'A'	10 'A' sto
->+	sto+	100 ->+ 'A'	100 'A' sto+
->-	sto-	100 ->- 'A'	100 'A' sto-
->*	sto*	100 ->* 'A'	100 'A' sto*
->/	sto/	100 ->/ 'A'	100 'A' sto/

Les tests avec 'if'

Le code qui effectue un test `if` est en réalité transformé par le parser de MOGWAI en un appel 100% RPN. Par exemple pour le test suivant :

```
if (A 100 ==) then {"A vaut 100" ?} else {"A ne vaut pas 100" ?}
```

MOGWAI va transformer cette ligne en :

```
{A 100 ==} eval {"A vaut 100" ?} {"A ne vaut pas 100" ?} IFELSE
```

Ce qui en pratique se traduit par cette série d'opérations :

1. { A 100 == } est évalué afin de poser sur la pile le résultat booléen du test à effectuer.
2. { "A vaut 100" ? } est posé sur la pile.
3. { "A ne vaut pas 100" ? } est posé sur la pile.
4. La fonction IFELSE pend la valeur booléenne, si elle est vraie elle évalue le 1er bloc de code, sinon elle évalue le second bloc de code.

Avouez que la version non RPN est plus sympa à écrire et à comprendre.

Les fonctions de manipulation de la pile

La pile peut être manipulée car dans certains cas c'est bien pratique. Cela évite souvent d'utiliser des variables locales intermédiaires. Au final le code est plus rapide.

Par exemple, si vous souhaitez effectuer un calcul, afficher le résultat, puis effectuer un autre calcul à partir de ce résultat et l'afficher aussi, théoriquement vous avez besoin d'une variable intermédiaire :

```
# On fait un 1er calcul et on l'affiche.
# Mais on doit garder une trace du résultat pour un autre calcul après.

# On fait le 1er calcul et on stocke le résultat dans A.
2 7 + -> 'A'

# On affiche le résultat du 1er calcul.
A ?

# On fait le second calcul à partir du résultat du calcul précédent qu'on affiche tout de suite.
A 200 * ?
```

En manipulant la pile on peut éviter la variable intermédiaire et rendre le code plus compact et plus rapide. Pour cela nous allons utiliser la fonction dup qui duplique le 1er élément de la pile :

```
# On fait le 1er calcul et on duplique le résultat pour l'afficher.
# Puis on fait le second calcul à partir du résultat du calcul précédent qu'on affiche.

2 7 + dup ?
200 * ?
```

Les fonctions de pile disponibles

dup	Duplique le 1er élément de la pile.
swap	Intervertit le 1er et le 2ème élément de la pile.
clear	Vide la pile.
depth	Pose sur la pile la taille de la pile au moment de la demande.
drop	Supprime le 1er élément de la pile.
pop	Prélève le 1er élément de la pile et le stocke pour la fonction push
push	Pose sur la pile l'objet stocké par la fonction pop

La fonction sign

Il est possible de déterminer le type des éléments de la pile sans pour autant dépiler les éléments. La fonction `sign` qui prend en paramètre le nombre d'éléments à inspecter retourne une liste contenant les types des éléments inspectés.

```
# On pose sur la pile 3 valeurs de types différents

10 "EE" (1 2)

# On inspecte ces 3 valeurs

3 sign

# sign pose sur la pile la liste (.list .string .number)
# Qui correspondent aux types des éléments présents sur la pile
# En position zéro dans la liste le type du dernier élément posé sur la pile
```

Si on essaye d'inspecter plus d'éléments que réellement présents sur la pile, la fonction `sign` retourne la valeur `null`.

La fonction `sign` est très utile pour vérifier, sans modifier la pile, que les paramètres présents sont bien du type attendu.

TESTS

L'instruction if

`if` permet d'effectuer des tests et de prendre de décisions.

Quand le test est positif un bloc de code est exécuté. Il est possible aussi de définir un bloc de code à exécuter quand le test est négatif.

```
50 -> 'A'

if (A 50 ==) then
{
    "A est égal à 50" ?
}
else
{
    "A n'est pas égal à 50" ?
}
```

Il est impératif que la clause de test (le code placé entre les parenthèses) pose sur la pile une valeur booléenne. Si ce n'est pas le cas une erreur est levée.

```
# Cette expression va fonctionner
if (true) then {"VRAI !" ?} else {"FAUX !" ?}


# Cette expression va lever une erreur
if ("TOTO") then {"VRAI !" ?} else {"FAUX !" ?}
```

Opérations logiques booléennes (retournent true ou false)

X Y ==	X égale à Y ?
X Y !=	X différent de Y ?
X Y >	X plus grand que Y ?
X Y >=	X plus grand ou égal à Y ?
X Y <	X plus petit que Y ?
X Y <=	X plus petit ou égal à Y ?
X not	Inversion logique de X
X Y or	X OU Y
X Y and	X ET Y
X Y xor	OU EXCLUSIF entre X et Y

Opérations logiques binaires (retournent un nombre)

X Y &	ET binaire
X Y |	OU binaire
X Y ^	OU EXCLUSIF BINAIRE
X Y ~	NON binaire

L'instruction switch

Pour éviter les `if .. else` en cascade vous pouvez utiliser l'instruction `switch`.

Cette instruction est composée de plusieurs couples test / bloc de code.

Au 1er test qui retourne `true` rencontré, son bloc de code est exécuté et uniquement lui.

```
# On veut afficher un message suivant la valeur de la variable 'a'

150 -> 'a'

switch
{
    (a 100 <)
    {
        "100" ?
    }

    (a 200 <)
    {
        "< 200" ?
    }

    (true)
    {
        "DEFAULT" ?
    }
}
```

Si l'on veut absolument avoir un bloc de code qui s'exécute même si aucun autre n'est sélectionné (une sorte de bloc par défaut), il suffit de mettre en dernier un bloc dont le test ne peut pas échouer (idéalement on place `true` directement dans le test).

BOUCLES

repeat

Pour exécuter un bloc de code un certain nombre de fois il faut utiliser `repeat`.

```
# On va afficher les nombres de 1 à 10
# La variable 'I' sert de compteur de boucle

0 -> 'I'

10 repeat
{
    'I' ++
    I ?
}

# On va afficher les nombres de 1 à 10
# La variable 'I' sert de compteur de boucle
# On sort de la boucle quand 'I' est égal à 5

0 -> 'I'

10 repeat
{
    'I' ++
    I ?

    if (I 5 ==) then {break}
}
```

during

Pour exécuter un bloc de code pendant une certaine durée il faut utiliser `during`. La durée est exprimée en millisecondes (1000 = 1 seconde).

```
# On va exécuter le code pendant 10 secondes

0 -> 'I'

during 10000 do
{
    'I' ++
    I ?
}
```

for

Pour utiliser un compteur de boucle géré automatiquement, il faut utiliser le mot clé `for`.

```
# On va afficher les nombres de 1 à 10
# La variable 'I' sert de compteur de boucle


1 10 for 'I' do
{
    I ?
}

# On va afficher les nombres de 10 à 1
# La variable 'I' sert de compteur de boucle


10 1 for 'I' step -1 do
{
    I ?
}

# On va afficher les nombres de 10 à 1
# La variable 'I' sert de compteur de boucle
# Quand on arrive à la valeur 5 on sort de la boucle


10 1 for 'I' step -1 do
{
    I ?

    if (I 5 ==) then {break}
}
```

foreach

Pour itérer chaque élément d'une liste il faut utiliser `foreach`.

```
# On affiche chaque élément de la liste

("LUN" "MAR" "MER" "JEU" "VEN" "SAM" "DIM") foreach 'jour' do {jour ?}
```

forever

Pour exécuter une boucle indéfiniment, il faut utiliser `forever`.

```
# On exécute le code suivant indéfiniment

0 -> 'I'

forever do {'I' ++ ?}

# On exécute le code suivant indéfiniment
# Mais on en sort quand 'I' a la valeur 456

0 -> 'I'

forever do
{
    'I' ++
    I ?

    if (I 456 ==) then {break}
}
```

while

Pour exécuter un bloc de code tant qu'une condition est vraie, il faut utiliser `while`. Avec cette écriture (`while` en début de boucle), le test est réalisé en premier.

```
# Tant que I est inférieur à 100 on l'affiche

0 -> 'I'

while (I 100 <) do
{
    'I' ++
    I ?
}
```

do... while

Pour exécuter un bloc de code tant qu'une condition est vraie, il faut utiliser `do … while`. Avec cette écriture, le code de la boucle est exécuté et le test est effectué à la fin.

```
# Tant que I est inférieur à 100 on l'affiche

0 -> 'I'

do
{
    'I' ++
    I ?
} while (I 100 <)
```

FONCTIONS MATHEMATIQUES

Fonction	Usage	Exemple
->deg	Converti un angle radian en degré.	0.05 ->deg
->rad	Converti un angle degré en radian.	3.14 ->rad
+	Ajoute 2 nombres.	5 7 +
-	Soustrait 2 nombres.	100 45 -
*	Multiplie 2 nombres.	10 60 *
/	Divise 2 nombres.	134 5 /
abs	Retourne la valeur absolue d'un nombre.	-56 abs
acos	Retourne l'arc cosinus d'un angle en radian.	0.5 acos
asin	Retourne l'arc sinus d'un angle en radian.	0.5 asin
atan	Retourne l'arc tangente d'un angle en radian.	0.5 atan
ceil	Retourne la valeur du plus petit entier supérieur ou égal au nombre spécifié.	56.89 ceil
cos	Retourne le cosinus d'un angle en radian.	0.5 cos
max	Retourne la valeur maximum d'une liste. Seuls les nombres sont pris en compte. Retourne null si la liste ne contient aucun nombre.	(1 5 2) max
mean	Retourne la moyenne d'une liste. Seuls les nombres sont pris en compte. Retourne 0 si la liste ne contient aucun nombre.	(10 20 30) mean
min	Retourne la valeur minimum d'une liste. Seuls les nombres sont pris en compte. Retourne null si la liste ne contient aucun nombre.	(1 5 2) min
pow	Retourne un nombre spécifié élevé à la puissance spécifiée.	100 2 pow
rand	Génère un nombre aléatoire compris entre 0 et 1.	rand

Fonction	Usage	Exemple
shift	Effectue un décalage de bits sur un nombre spécifié. Si la valeur de décalage est positive le décalage est effectué vers la droite, sinon vers la gauche.	100 4 shift
sin	Retourne le sinus d'un angle en radian.	0.5 sin
sqrt	Retourne la racine carrée d'un nombre.	16 sqrt
sum	Retourne la somme d'une liste. Retourne null si la liste ne contient aucun nombre.	(1 5 7) sum
tan	Retourne la tangente d'un angle en radian.	0.5 tan
PI	Retourne PI en degré.	PI
floor	Retourne la plus grande valeur intégrale inférieure ou égale au nombre spécifié.	45.8 floor
mod	Retourne le reste de la division entière d'un nombre par un autre.	100 3 mod

CHAINES DE CARACTERES (STRING)

MOGWAI possède de nombreuses fonctions de traitement des chaînes de caractères.

Concaténation

La fonction + permet de concaténer 2 chaînes de caractères. Cette fonction possède une certaine « intelligence » car suivant le contexte elle sait s'adapter.

La règle de base est que si le 1er argument est une chaîne alors quelques soit le second il sera transformé en chaîne de caractères et concaténé avec le 1er argument.

Opération	Résultat
"HELLO" "LE MONDE" +	"HELLO LE MONDE"
"HELLO" 3 +	"HELLO3"
3 "HELLO" +	"3HELLO"

Extraction

Il existe plusieurs fonctions pour extraire une partie d'une chaîne de caractères.

Opération	Résultat
"HELLO LE MONDE" 0 5 sub	"HELLO"
"HELLO LE MONDE" butfirst	"ELLO LE MONDE"
"HELLO LE MONDE" butlast	"HELLO LE MOND"
"HELLO LE MONDE" first	"H"
"HELLO LE MONDE" last	"E"

Taille

Pour récupérer la taille d'une chaîne de caractères il faut utiliser la fonction `size`.

```
# On récupère la taille d'une chaîne de caractères et on l'affiche

"HELLO LE MONDE" size ?
```

Trouver des éléments

Pour rechercher une sous-chaîne dans une chaîne de caractères il faut utiliser la fonction `where` qui retourne une liste composée de toutes les positions correspondantes.

```
# On recherche l'emplacement de toutes les lettres "E"

"HELLO LE MONDE" "E" where

# La réponse sera la liste (1 7 13)
```

Transformations

Pour transformer une chaîne de caractères vous pouvez utiliser les fonctions suivantes :

Opération	Résultat
"HELLO LE MONDE" ->lower	"hello le monde"
"hello le monde" ->upper	"HELLO LE MONDE"
("X" "Y" "Z") ";" join	"X;Y;Z"
"X;Y;Z" ";" split	("X" "Y" "Z")
"HELLO LE MONDE" reverse	"EDNOM EL OLLEH"

Formatage d'un nombre

Il est possible de formater un nombre grâce à la fonction ->format qui prend en paramètres le nombre à formater et le format à appliquer.

Le format à appliquer est une chaine de caractères décrivant quelle forme doit prendre le nombre :

Opération	Résultat
50.678 "0.00" ->format	"50.68"
34 "000" ->format	"034"

```
# On affiche la date courante au format dd/mm/yyyy
# Voir le chapitre GESTION DES DATES pour comprendre pleinement le code suivant.

now ->date -> 'dt'

dt day: get "00" ->format ??
"/" ??
dt month: get "00" ->format ??
"/" ??
dt year: get "0000" ->format ?
```

LES FONCTIONS DE CONVERSION

Pour convertir un objet en un autre (par exemple une chaîne de caractères en nombre ou l'inverse) MOGWAI dispose de fonctions de conversion qui commencent par le symbole ->.

Opération	Résultat
DATA:4142434445 ->ascii	"ABCDE"
DATA:4142434445 ->ascii7	"ABCDE"
45 ->str	"45"
"45" ->num	45
DATA:FF5612AE5678 ->base64	"/1YSrlZ4"
"/1YSrlZ4" ->base64	DATA:FF5612AE5678
1968 ->bin	BIN:11110110000
(64 65 66) ->data	DATA:414243
64 65 66 3 ->data	DATA:414243
0.56 ->deg	32.08563652732611
123.67432 "0.00" ->format	"123.67"
234 ->hex	"EA"
20 ->i8	DATA:14
20 ->i16	DATA:0014
20 ->i32	DATA:00000014
20 ->i64	DATA:0000000000000014
-30 ->u8	DATA:E2
-30 ->u16	DATA:FFE2
-30 ->u32	DATA:FFFFFFE2
-30 ->u64	DATA:FFFFFFFFFFFFFFE2
56.9865 ->int	56
"latitude" ->key	latitude:
"rand" ->keyword	rand

Opération	Résultat
45 56 78 3 ->list	(45 56 78 3)
DATA:414243 ->list	(65 66 67)
"HELLO LE MONDE" ->lower	"hello le monde"
"hello le monde" ->upper	"HELLO LE MONDE"
DATA:2345E323 ->md5	DATA:0E9751A0F9AF52C737038B4F2108A907
"latitude" ->name	'latitude'
(2 3 +) ->program	« 2 3 + »
35.3 ->rad	0.6161012259539983
DATA:12ED45FE89 ->sha1	DATA:8B1FB372469A9B52DED84498FF26CEE06C07910B
123 ->type	.number
(1 2 3) ->type	.list
'latitude' ->type	.name
latitude: ->type	.key
"latitude" ->type	.string
"Hello !" ->utf8	DATA:48656C6C6F2021
DATA:48656C6C6F2021 ->utf8	"Hello !"

LES LISTES (LIST)

Les listes de MOGWAI ne sont pas typées, elles peuvent contenir une collection de n'importe quels objets.

Les listes sont notées avec des parenthèses. Les objets qu'elles contiennent sont simplement séparés par des espaces.

Par exemple (1 2 7) est une liste de nombres, ("JE" "TU" "IL") est une liste de chaînes de caractères et ("JE" "TU" "IL" 45 67 (1 2 3) true) est une liste de pleins d'objets différents (vous remarquerez qu'une liste peut contenir des listes).

Créer une liste

La méthode la plus simple pour créer une liste est de la saisir directement (comme au-dessus).

Vous pouvez aussi poser sur la pile les éléments qui doivent la composer, indiquer combien il faut en prendre et utiliser la fonction `->list`.

```
# On crée une liste à partir des objets qui sont sur la pile.

10 20 30 40 50 5 ->list

# Cette instruction va poser sur la pile la liste (10 20 30 40 50)
```

Vous pouvez aussi taper directement la liste dans votre code :

```
# On crée une liste directement dans le code

(10 20 30 40 50)

# Cette instruction va poser sur la pile la liste (10 20 30 40 50)
```

Ajouter des éléments à une liste

La fonction `+` permet d'ajouter un élément à une liste.

```
# On ajoute 1 élément à une liste.

(10 20 30) 40 +

# Cette instruction va poser sur la pile la liste (10 20 30 40)

# On ajoute une liste à une liste.

(10 20 30) (100 200) +

# Cette instruction va poser sur la pile la liste (10 20 30 (100 200))
```

Récupérer la taille (nombre d'éléments) d'une liste

La fonction `size` retourne la taille d'une liste.

```
# On récupère la taille d'une liste pour l'afficher

(10 20 30 40) size ?

# Affichera 4
```

Modifier un élément d'une liste

La fonction `set` permet de modifier un élément particulier. On doit fournir son index (de 0 à size-1) et la nouvelle valeur :

```
# On modifie le 3ème élément de la liste (on remplace 55 par "Z")

(10 "E" 55 20 30) 2 "Z" set

# Cette instruction va poser sur la pile la liste (10 "E" "Z" 20 30)
```

Récupérer un élément d'une liste

La fonction `get` permet de récupérer un élément d'une liste. Comme avec `set`, on doit fournir son index (de 0 à size-1) :

```
# On récupère le 5ème élément de la liste

(10 20 30 40 50 60 70) 5 get

# Cette instruction va poser sur la pile la valeur 60
```

Si l'index spécifié n'est pas dans la plage possible (de 0 à size-1) la fonction retourne `null` et ne lève pas d'erreur.

Récupérer un élément "enfoui" dans une liste

Si une liste est composée, de sous-listes et/ou de sous record (voir plus loin la présentation des records qui sont des associations clé/valeur) il peut être intéressant de donner en une seule opération le "chemin" à suivre pour récupérer l'information :

```
# Méthode 1, basique, on récupère une information en plusieurs opérations
# On va d'abord récupérer le 2ème record, puis la valeur de sa clé name:

([id: 0 name: "MARTIN"] [id: 1 name: "DUPONT"] [id: 2 name: "DURANT"]) 1 get

# Cette opération pose sur la pile le record [id: 1 name: "DUPONT"]
# Puis on récupère la valeur de la clé name:

name: get

# Ce qui pose sur la pile "DUPONT"
```

```
# Méthode 2, on récupère une information en une seule opération

([id: 0 name: "MARTIN"] [id: 1 name: "DUPONT"] [id: 2 name: "DURANT"]) (1 name:) get

# Cette opération pose directement sur la pile "DUPONT"
```

Si le chemin abouti à rien (mauvais chemin) la valeur retournée sera la valeur null.

```
# Si le chemin est mauvais

([id: 0 name: "MARTIN"] [id: 1 name: "DUPONT"] [id: 2 name: "DURANT"]) (5 name:) get

# Cette opération pose directement sur la pile null car l'élément 5 de la liste
# N'existe pas.
```

Extraire une partie d'une liste

La fonction `extract` permet d'extraire que certains éléments d'une liste en une seule opération. Elle prend en paramètres la liste source et une liste des indexes à extraire :

```
# On extrait les éléments 1 2 4 de la liste

(10 "E" 55 20 30) (1 2 4) extract

# Cette instruction va poser sur la pile la liste ("E" 55 30)
```

Si on demande des indexes qui n'existent pas (en dehors des indexes de la liste source) les valeurs de type null seront ajoutée à leur place.

Récupérer le 1er élément d'une liste

Il existe 2 manières, la 1ère est celle que nous venons de voir, en utilisant la fonction `get` avec un index valant zéro.

La seconde manière est d'utiliser la fonction `first`, qui fait exactement la même chose. Si la liste est vide, elle retourne `null`.

```
# On récupère le 1er élément de la liste de 2 manières

# Avec la fonction get

(10 20 30 40 50 60 70) 0 get

# Cette instruction va poser sur la pile la valeur 10

# Avec la fonction first

(10 20 30 40 50 60 70) first

# Cette instruction va poser sur la pile la valeur 10
```

Récupérer le dernier élément d'une liste

Vous l'aurez sans doute deviné, la fonction `last` retourne le dernier élément d'une liste, et la valeur `null` si la liste est vide.

```
# On récupère le dernier élément de la liste

(10 20 30 40 50 60 70) last

# Cette instruction va poser sur la pile la valeur 70
```

Supprimer un élément d'une liste

Pour supprimer un élément d'une liste il faut utiliser la fonction `purge` avec comme paramètres la liste et l'index à supprimer. Si l'index est < 0 une erreur est levée. Si l'index est >= size l'opération est simplement ignorée.

```
# On supprime le 3ème élément de la liste, soit la valeur 40

(10 20 30 40 50 60 70) 3 purge

# Cette instruction va poser sur la pile (10 20 30 50 60 70)
```

Extraire les éléments d'une liste à partir d'un index donné

Pour extraire une sous-liste il faut utiliser la fonction `sub` avec comme paramètres l'index de début et le nombre d'éléments à récupérer. Si l'index de début est en dehors de la liste une erreur est levée. Cette fonction retourne une liste composée des éléments sélectionnés. Si on demande plus d'éléments que possible la réponse sera composée du maximum d'éléments possibles.

```
# On récupère une partie d'une liste

(10 20 30 40 50 60 70) 2 3 sub

# Cette instruction va poser sur la pile (30 40 50)

# On récupère une partie d'une liste en demandant trop d'éléments

(10 20 30 40 50 60 70) 2 30 sub

# Cette instruction va poser sur la pile (30 40 50 60 70)

# On récupère une partie d'une liste en partant d'un index trop grand

(10 20 30 40 50 60 70) 20 3 sub

# Cette instruction va lever l'erreur "bad argument value"
```

Récupérer toute une liste sauf le 1er élément ou le dernier élément

C'est la fonction `butfirst` qui permet de récupérer toute une liste sauf le 1er élément. Et la fonction `butlast` permet de récupérer toute une liste sauf le dernier élément. Si la liste est vide ou si elle se compose d'un seul élément, ces fonctions retournent une liste vide.

```
# On récupère une liste sans son 1er élément

(10 20 30 40 50 60 70) butfirst

# Cette instruction va poser sur la pile (20 30 40 50 60 70)

# On récupère une liste sans son dernier élément

(10 20 30 40 50 60 70) butlast

# Cette instruction va poser sur la pile (10 20 30 40 50 60)
```

Convertir une liste en tableau de bytes (data)

Vous pouvez créer un objet `data` (tableau d'octets) à partir d'une liste. Seuls les nombres compris entre 0 et 255 seront pris en compte et intégrés au résultat final.

```
# Exemple 1 : On crée un objet data à partir d'une liste d'octets exprimés en hexadécimal

(0x10 0x20 0x30 0x40) ->data

# Cette instruction pose sur la pile l'objet data DATA:10203040

# Exemple 2 : On n'est pas obligé d'utiliser la notation hexadécimale

(100 200 3000 "EEE" 120 10 true) ->data
```

```
# Cette instruction va poser sur la pile l'objet data DATA:64C8B8780A
# Les éléments 3000 "EEE" true ont été ignorés car ce ne sont pas des octets.
```

Trouver l'emplacement de valeurs

Pour rechercher l'emplacement de valeurs dans une liste il faut utiliser la fonction `where`. Cette fonction retourne tous les emplacements d'une valeur qui lui est passée en paramètre.

```
# On cherche les indexes de la valeur "XX"

(10 20 "XX" "EA" 670 true "XX") "XX" where

# Cette instruction va poser sur la pile (2 6)
```

Vérifier qu'une valeur est présente au moins une fois dans une liste

La fonction `contains` retourne une valeur booléenne indiquant si une valeur est présente (au moins une fois) dans une liste.

```
# On vérifie que la valeur "JEU" est présente dans la liste

("LUN" "MAR" "MER" "JEU" "VEN" "SAM" "DIM") "JEU" contains

# Cette instruction va poser sur la pile true
```

Fonctions mathématiques

Quelques fonctions mathématiques utilisent des listes comme paramètre d'entrée. C'est le cas par exemple des fonctions `sum`, `mean`, `min`, `max`.

Le paragraphe "Fonctions mathématiques" explique leur utilisation.

LES ENREGISTREMENTS (RECORD)

Les enregistrements MOGWAI sont des objets permettant d'associer une valeur à une clé (un peu comme un dictionnaire).

L'objet clé

La clé d'une association est confiée à un objet key qui est un nom finissant obligatoirement par le symbole ":" (deux points).

L'objet RECORD

Un objet record est délimité par des crochets [ ] et contient une série de clés/valeurs. Un record peut être vide, dans ce cas il est noté juste [ ].

Par exemple, un record contenant une valeur x et y aura une clé `x:` et une clé `y:` et leur valeur, ce qui donnerai : `[x: 100 y: 50].`

La valeur peut être n'importe quel objet MOGWAI, et pourquoi pas une clé (qui est un objet MOGWAI donc autorisé), ou un autre record.

Une clé ne peut être présente qu'une seule fois dans un record. Si ce n'est pas le cas, seule la valeur de la dernière occurrence de la clé est prise en compte.

`[x: 10 y: 20 x: 100]` revient à écrire `[x: 100 y: 20]`

Ajouter ou modifier des clés

Pour ajouter une nouvelle clé ou modifier une clé existante il faut utiliser la fonction `set` en lui stipulant le record à traiter, la clé à utiliser et la valeur associée.

```
# Exemple 1 : On ajoute la clé z: avec la valeur 300

[x: 100 y: 200] z: 300 set

# Cette instruction pose sur la pile [x: 100 y: 200 z: 300]

# Exemple 2 : On modifie la clé y: en lui donnant la valeur 2000 à la place de 200

[x: 100 y: 200] y: 2000 set

# Cette instruction pose sur la pile [x: 100 y: 2000]
```

Récupérer la valeur d'une clé

Pour récupérer la valeur d'une clé il faut utiliser la fonction `get` en lui indiquant le record et la clé.

```
# On ajoute récupère la valeur de la clé y:

[x: 100 y: 200] y: get

# Cette instruction pose sur la pile 200
```

Récupérer une clé "enfouie" dans un record

Si un record est composé, de sous-records et/ou de sous listes il peut être intéressant de donner en une seule opération le "chemin" à suivre pour récupérer l'information :

```
# Méthode 1, basique, on récupère une information en plusieurs opérations
# On va d'abord récupérer la valeur de la clé gps: puis la valeur de la clé latitude:

[id: 1 name: "DUPONT" gps: [latitude: 45 longitude: 5]] gps:

# Cette opération pose sur la pile le record [latitude: 45 longitude: 5]
# Puis on récupère la valeur de la clé latitude:

latitude: get

# Ce qui pose sur la pile 45
```

```
# Méthode 2, on récupère l'information en une seule opération

[id: 1 name: "DUPONT" gps: [latitude: 45 longitude: 5]] (gps: latitude:) get

# Cette opération pose directement sur la pile 45
```

Récupérer la taille d'un record (nombre de clés)

La fonction `size` retourne le nombre de clés présentes dans un record.

```
# On récupère le nombre de clés du record

[x: 100 y: 200] size

# Cette instruction pose sur la pile 2
```

Récupérer la liste des clés d'un record

La fonction `keys` retourne la liste des clés d'un record.

```
# On récupère la liste des clés d'un record

[x: 100 y: 200] keys

# Cette instruction pose sur la pile (x: y:)
```

Extraire une partie d'un record

La fonction `extract` permet d'extraire que certaines clés d'un record en une seule opération. Elle prend en paramètres le record source et une liste des clés à extraire :

```
# On extrait les clés x: y: du record

[x: 100 y: 200 z: 70 u: 10] (x: y:) extract

# Cette instruction va poser sur la pile le record [x: 100 y: 200]
```

Si on demande une clé qui n'existe pas elle est ajoutée au record résultat avec la valeur null.

Vérifier qu'une clé est présente dans un record

La fonction `contains` retourne une valeur booléenne indiquant si une clé est dans une liste.

```
# On vérifie que la clé x: est présente dans un record

[x: 10 y: 20] y: contains

# Cette instruction va poser sur la pile true
```

Supprimer une clé dans un record

La fonction `purge` permet de supprimer une clé. Elle prend en paramètres le record et la clé à supprimer.

```
# On supprime la clé x: du record

[x: 10 y: 20] x: purge

# Cette instruction va poser sur la pile [y: 20]
```

Notation plus « courte » des clés

MOGWAI autorise une notation plus compacte pour les clés passées en paramètre lors de l'utilisation des foncions `get` et `set` en utilisant les symboles `->` et `<-`.

Cette notation est acceptée uniquement avec un nom de variable, pas directement avec un record :

```
# On place un record dans une variable et on récupère la valeur de la clé y

[x: 10 y: 20] -> 'A'

A->y ?

# Cette instruction pose sur la pile la valeur 20
# Cette instruction est la version compacte de A y: get ?
```

```
# On place un record dans une variable et on lui ajoute une clé

[x: 10 y: 20] -> 'A'

500 A<-z

# Cette instruction pose sur la pile [x: 10 y: 20 z: 500]
# Cette instruction est la version compacte de A z: 500 set
# Attention la variable A n'est pas modifiée.
```

LES TABLEAUX D'OCTETS (DATA)

Dans le domaine industriel, il est très souvent nécessaire de manipuler des tableaux d'octets. On envoie des commandes sous la forme de tableaux d'octets, on reçoit des informations sous la même forme. Il est souvent question de manipuler ces données un peu dans tous les sens.

MOGWAI ayant à la base été créé pour simuler un appareil utilisant le Bluetooth Low Energy, il possède naturellement toute une batterie de fonctions pour manipuler le plus simplement possible des tableaux d'octets et les octets eux même.

Un tableau d'octet est nommé `DATA` en MOGWAI et le type est `.data`

Il est possible de créer un `DATA` directement avec la notation `DATA:` suivie des octets qui le composent au format hexadécimal :

```
# On crée un tableau d'octets composé de 4 octets
# Qui sont AB 56 32 FF

DATA:AB5632FF

# Pose sur la pile le tableau de 4 octets
```

On peut aussi créer un DATA vide avec la fonction `new` :

```
# On crée un DATA vide et on le stocke
# Dans la variable globale $D

.data new -> '$D'
```

On peut ajouter un octet au `DATA` avec la fonction `+` :

```
# On crée un tableau d'octets composés de 4 octets
# Qui sont 0xAB 0x56 30x2 0xFF

DATA:AB5632FF

# Il est posé sur la pile
# On lui ajoute maintenant un octet de valeur 0x56

0x56 +

# Sur la pile il y a maintenant DATA:AB5632FF56
```

La fonction `size` retourne la taille (le nombre d'octets) du `DATA`.

On peut concaténer 2 DATA avec la fonction + :

```
# On place 2 DATA dans 2 variables globales

DATA:FF56EB23 -> '$A'
DATA:89CD34 -> '$B'

# On concatène les 2 DATA qu'on stocke dans une autre variable globale

$A $B + -> '$C'

# $C contient maintenant DATA:FF56EB2389CD34
```

Pour récupérer un octet particulier d'un DATA il faut utiliser la fonction get (le 1er octet possède l'index zéro) :

```
# On crée un DATA composé de 4 octets et on
# Extrait l'octet placé en 3ème position

DATA:FF56EB23 2 get

# La valeur 0xEB (235 en décimal) est posée sur la pile
```

Pour modifier la valeur d'un octet particulier, il faut utiliser la fonction set :

```
# On crée un DATA composé de 4 octets
# Puis on modifie l'octet placé en position 1
# La valeur 0x56 sera remplacée par 0x34

DATA:FF56EB23 1 0x34 set

# DATA:FF34EB23 est posé sur la pile
```

Pour modifier une partie d'un DATA par un autre DATA, il faut aussi utiliser la fonction set :

```
# On va remplacer les 2 premiers octets d'un DATA

DATA:FFC0AB0146 0 DATA:AABB set ?

# Il y a maintenant sur la pile DATA:AABBAB0146
```

Pour supprimer un octet particulier, il faut utiliser la fonction purge :

```
# On crée un DATA composé de 4 octets
# Puis on supprime l'octet placé en position 1

DATA:FF56EB23 1 purge

# DATA:FFEB23 est posé sur la pile
```

Pour extraire une partie d'un DATA il faut utiliser la fonction sub :

```
# On crée un data composé de 6 octets
# On extrait à partir du 3ème octet, 3 octets

DATA:010203EB5634 2 3 sub

# DATA:03EB56 est posé sur la pile
```

La fonction extract permet d'extraire que certains éléments d'un data en une seule opération. Elle prend en paramètres le data source et une liste des indexes à extraire :

```
# On extrait les éléments 1 2 4 du data

DATA:FF45AB23EA (1 2 4) extract

# Cette instruction va poser sur la pile le data DATA:45ABEA
```

Si on demande des indexes qui n'existent pas (en dehors des indexes du data source) une erreur est levée.

Il est possible de transformer un DATA en une liste de nombres avec la fonction ->list :

```
# On transforme un DATA en liste

DATA:FF45EB12AD89 ->list

# La liste (255 69 235 18 173 137) est posée sur la pile
```

A partir d'une liste de nombres on peut créer un DATA avec la fonction ->data. Attention, seuls les nombres compris entre 0 et 255 seront pris en compte, les autres éléments de la liste seront ignorés :

```
# On transforme une liste en DATA

(50 25 45 36 0xFF) ->data

# DATA:32192D24FF est posé sur la pile
```

Toujours avec la fonction ->data, Il est possible de créer un DATA directement à partir des éléments placés sur la pile. Il suffit d'indiquer combien d'éléments sont à utiliser. Attention, les éléments qui ne sont pas des nombres ou dont la valeur n'est pas comprise entre 0 et 255 sont ignorés :

```
# On transforme les éléments de la pile en DATA
# On doit indiquer combien d'éléments sont à utiliser
# Ici 6

50 25 45 36 12 0xFF 6 ->data ?

# DATA:32192D240CFF est posé sur la pile
```

Pour trouver toutes les occurrences d'un octet dans un DATA, il faut utiliser la fonction where :

```
# On va rechercher toutes les occurrences de la valeur 0xC0 dans un DATA

DATA:FFC005FA12C056EC 0xC0 where

# where va poser sur la pile la liste (1 5)
# Car dans ce DATA, la valeur 0xC0 est présente à la position 1 et 5
```

On peut aussi trouver les emplacements d'un DATA dans un autre :

```
# On va rechercher toutes les occurrences de 0xFFC0 dans un DATA
# Cela revient à chercher un DATA dans un autre (ici DATA:FFC0)

DATA:FFC005FA12C056EC DATA:FFC0 where

# where va poser sur la pile la liste (0)
# Car dans ce DATA, la valeur 0xFFC0 est présente à la position 0 uniquement
```

Les fonctions de conversion vers un DATA

Pour manipuler efficacement des tableaux d'octets, il faut être en mesure de convertir des nombres dans différents formats. Par exemple, prendre un nombre et le convertir en un entier non signé sur 16 bits (2 octets), ou sur un entier signé sur 32 bits (4 octets) suivant les besoins.

MOGWAI propose à cet effet une série de fonctions de conversion qui prennent en paramètre un nombre et qui retournent le `DATA` correspondant après conversion. Par exemple, après la conversion d'un nombre en un entier signé sur 32 bits, on obtiendra un `DATA` composé des 4 octets correspondants au résultat de la conversion demandée.

Une fois la conversion terminée, il est assez simple d'insérer le résultat (qui est un `DATA`) dans un `DATA` avec la fonction `set`.

Fonctions de conversion de nombres retournant un `DATA` :

`50 ->u8`	`Conversion en entier non signé sur 8 bits (1 octet)` `DATA:32`
`50 ->u16`	`Conversion en entier non signé sur 16 bits (2 octets)` `DATA:0032`
`50 ->u32`	`Conversion en entier non signé sur 32 bits (4 octets)` `DATA:00000032`
`50 ->u64`	`Conversion en entier non signé sur 64 bits (8 octets)` `DATA:0000000000000032`
`-50 ->i8`	`Conversion en entier signé sur 8 bits (1 octet)` `DATA:CE`
`-50 ->i16`	`Conversion en entier signé sur 16 bits (2 octets)` `DATA:FFCE`
`-50 ->i32`	`Conversion en entier signé sur 32 bits (4 octets)` `DATA:FFFFFFCE`
`-50 ->i64`	`Conversion en entier signé sur 64 bits (8 octets)` `DATA:FFFFFFFFFFFFFFCE`

Si un nombre trop grand ou trop petit est fourni en paramètre, il sera tronqué pendant la conversion sans lever d'erreur.

Affichage évolué d'un DATA

Pour visualiser plus simplement le contenu d'un DATA, on peut utiliser la fonction ?dump (ou ?d) qui va afficher le dump d'un DATA.

```
# On va télécharger la page principale de google.fr
# On stocke la réponse dans la variable locale R

[uri: "https://www.google.fr"] http.get -> 'R'

# La fonction retourne un record composé de 2 clés
# state: de type .boolean qui indique si tout s'est bien passé
# response: de type .data qui contient la ressource téléchargée

if (R state: get) then
{
	# Tout est ok on peut récupérer la réponse
	# Qu'on stocke dans la variable locale B

	R response: get -> 'B'

	# On extrait les 1000 premiers octets et on les affiche
	# Sous la forme d'un dump

	B 0 1000 sub ?dump
}
```

Voici un exemple d'affichage du dump d'un DATA :

```
00000000  3C 21 64 6F 63 74 79 70 65 20 68 74 6D 6C 3E 3C  | <!doctype html>< |
00000010  68 74 6D 6C 20 69 74 65 6D 73 63 6F 70 65 3D 22  | html itemscope=" |
00000020  22 20 69 74 65 6D 74 79 70 65 3D 22 68 74 74 70  | " itemtype="http |
00000030  3A 2F 2F 73 63 68 65 6D 61 2E 6F 72 67 2F 57 65  | ://schema.org/We |
00000040  62 50 61 67 65 22 20 6C 61 6E 67 3D 22 66 72 22  | bPage" lang="fr" |
00000050  3E 3C 68 65 61 64 3E 3C 6D 65 74 61 20 63 6F 6E  | ><head><meta con |
00000060  74 65 6E 74 3D 22 74 65 78 74 2F 68 74 6D 6C 3B  | tent="text/html; |
00000070  20 63 68 61 72 73 65 74 3D 55 54 46 2D 38 22 20  |  charset=UTF-8"  |
00000080  68 74 74 70 2D 65 71 75 69 76 3D 22 43 6F 6E 74  | http-equiv="Cont |
00000090  65 6E 74 2D 54 79 70 65 22 3E 3C 6D 65 74 61 20  | ent-Type"><meta  |
000000A0  63 6F 6E 74 65 6E 74 3D 22 2F 69 6D 61 67 65 73  | content="/images |
000000B0  2F 62 72 61 6E 64 69 6E 67 2F 67 6F 6F 67 6C 65  | /branding/google |
000000C0  67 2F 31 78 2F 67 6F 6F 67 6C 65 67 5F 73 74 61  | g/1x/googleg_sta |
000000D0  6E 64 61 72 64 5F 63 6F 6C 6F 72 5F 31 32 38 64  | ndard_color_128d |
000000E0  70 2E 70 6E 67 22 20 69 74 65 6D 70 72 6F 70 3D  | p.png" itemprop= |
000000F0  22 69 6D 61 67 65 22 3E 3C 74 69 74 6C 65 3E 47  | "image"><title>G |
00000100  6F 6F 67 6C 65 3C 2F 74 69 74 6C 65 3E 3C 73 63  | oogle</title><sc |
00000110  72 69 70 74 20 6E 6F 6E 63 65 3D 22 66 4C 6F 78  | ript nonce="fLox |
00000120  59 71 79 59 4B 73 59 35 69 6E 59 78 79 4E 4F 4C  | YqyYKsY5inYxyNOL |
00000130  6E 41 22 3E 28 66 75 6E 63 74 69 6F 6E 28 29 7B  | nA">(function(){ |
00000140  76 61 72 20 5F 67 3D 7B 6B 45 49 3A 27 47 72 39  | var _g={kEI:'Gr9 |
00000150  62 61 50 58 77 45 2D 79 59 6B 64 55 50 5F 39 79  | baPXwE-yYkdUP_9y |
00000160  43 32 51 59 27 2C 6B 45 58 50 49 3A 27 30 2C 32  | C2QY',kEXPI:'0,2 |
00000170  30 32 37 39 32 2C 36 32 2C 32 2C 36 30 39 36 32  | 02792,62,2,60962 |
00000180  35 2C 33 38 38 2C 32 38 38 37 34 31 34 2C 31 31  | 5,388,2887414,11 |
00000190  30 31 2C 35 35 32 37 37 32 2C 34 32 35 36 30 33  | 01,552772,425603 |
000001A0  2C 32 34 37 33 31 39 2C 34 32 37 32 35 2C 35 32  | ,247319,42725,52 |
000001B0  33 30 32 38 30 2C 31 31 34 30 32 2C 33 32 37 36  | 30280,11402,3276 |
000001C0  38 39 33 33 2C 34 30 34 33 37 30 39 2C 32 35 32  | 8933,4043709,252 |
000001D0  32 38 36 38 31 2C 31 33 38 32 36 38 2C 31 34 31  | 28681,138268,141 |
000001E0  31 38 2C 31 31 39 34 30 2C 35 33 32 32 32 2C 36  | 18,11940,53222,6 |
```

DATA et chaînes de caractères

Certaines fonctions de conversion liées aux chaînes de caractères prennent des DATA en paramètres ou retournent des DATA :

DATA:414243 ->ascii	Retourne la chaîne de caractères ASCII (8 bits) composée avec les octets du DATA passé en paramètre. "ABC"
DATA:414243 ->ascii7	Retourne la chaîne de caractères ASCII (7 bits) composée avec les octets du DATA passé en paramètre. "ABC"
DATA:414243 ->utf8	Retourne la chaîne de caractères UTF8 composée avec les octets du DATA passé en paramètre. "ABC"
DATA:414243 ->base64	Retourne le tableau d'octet sous la forme d'une chaîne de caractères codée en base 64. "QUJD"
"ABC" ->ascii	Retourne le tableau d'octets correspondant à la conversion ASCII (8 bits) d'une chaîne de caractères. DATA:414243
"ABC" ->ascii7	Retourne le tableau d'octets correspondant à la conversion ASCII (7 bits) d'une chaîne de caractères. DATA:414243
"ABC" ->utf8	Retourne le tableau d'octets correspondant à la conversion UTF8 d'une chaîne de caractères. DATA:414243
"QUJD" ->base64	Retourne le tableau d'octets correspondant au décodage d'une chaîne codée en base 64. DATA:414243

Autres fonctions disponibles

Les fonctions de calcul de clés de hachage :

DATA:414243 ->md5	Retourne la clé de hachage MD5 d'un DATA DATA:902FBDD2B1DF0C4F70B4A5D23525E932
DATA:414243 ->sha1	Retourne la clé de hachage SHA1 d'un DATA DATA:3C01BDBB26F358BAB27F267924AA2C9A03FCFDB8

Il est possible, avec la fonction `compress` de compresser un `DATA`, et le décompresser avec la fonction `decompress` :

```
# On va télécharger la page principale de google.fr
# On stocke la réponse dans la variable locale R

[uri: "https://www.google.fr"] http.get -> 'R'

# La fonction retourne un record composé de 2 clés
# state: de type .boolean qui indique si tout s'est bien passé
# response: de type .data qui contient la ressource téléchargée

if (R state: get) then
{
	# Tout est ok on peut récupérer la réponse
	# Qu'on stocke dans la variable locale B
	# Et on compresse la réponse qu'on stocke dans la variable locale C

	R response: get -> 'B'

	B compress -> 'C'

	# On peut afficher la différence de taille

	B size ?
	C size ?

	# On décompresse C et on affiche sa taille

	C decompress size ?
}
```

LES NOMBRES BINAIRES (BIN)

Pour simplifier la manipulation des bits d'un nombre, il est possible d'utiliser un objet MOGWAI de type `.binary`.

Dans MOGWAI, un nombre binaire commence par `BIN:` suivi des bits utilisés. Par exemple le nombre binaire 11001101 en binaire s'écrit dans MOGWAI `BIN:11001101`.

On ne peut pas gérer un nombre binaire de plus de 64 bits.

La fonction size retourne la taille (en bits) du nombre binaire.

Il est possible de créer un nombre binaire avec la fonction `new`. Par défaut `new` crée un nombre binaire de 8 bits.

```
# On crée un nombre binaire avec new

.binary new

# Ce qui pose BIN:00000000 sur la pile
# On aurait pu directement écrire BIN:00000000 aussi

# On affiche maintenant la taille en bits de ce nombre binaire
# La valeur affichée sera 8

size ?
```

Avec la fonction `resize`, on peut à tout moment modifier la taille d'un nombre binaire pour l'adapter aux besoins :

```
# On crée un binaire sur 8 bits

BIN:11000111

# On le passe en 16 bits

16 resize

# On affiche sa taille qui est 16 bits maintenant

size ?
```

Il est possible d'assembler 2 nombres binaires avec la fonction `+` :

```
# On assemble 2 nombres binaires
# Le 1er fait 1 bit, et le second 7 bits
# Le tout fera donc 8 bits au final

BIN:1 BIN:1111111 +

# Pose sur la pile BIN:11111111
```

Avec la fonction `->bin`, on peut créer un nombre binaire à partir d'un nombre classique. Le nombre de bits du nombre binaire créé sera limité à ceux nécessaires à la représentation du nombre d'origine.

Par exemple, le nombre 112 en binaire s'écrit 1110000, le nombre binaire créé a donc une taille de 7 bits. Pour travailler sur un nombre plus standard de bits (ex 16 bits) il faut utiliser dans la foulée la fonction `resize` :

```
# On veut créer un nombre binaire sur 16 bits
# Ayant pour valeur 112

112 ->bin 16 resize

# Pose sur la pile BIN:0000000001110000
```

La fonction `up` permet de lever un bit donné, et la fonction `down` permet le contraire. Il faut donner à ces fonctions le numéro du bit à modifier (le 1er bit a pour numéro 0) :

```
# On crée un nombre binaire sur 16 bits ayant
# La valeur 112

112 ->bin 16 resize

# Pose sur la pile BIN:0000000001110000

# On lève le bit 15 et on tombe le bit 4

15 up
5 down

# Pose sur la pile BIN:1000000001010000
```

Pour extraire une partie d'un nombre binaire il faut utiliser la fonction `sub` en indiquant à partir de quel bit on effectue l'extraction et combien de bits extraire. La fonction retourne un nombre binaire composé des bits extraits :

```
# On crée un nombre binaire sur 16 bits ayant
# La valeur 112

112 ->bin 16 resize

# Pose sur la pile BIN:0000000001110000

# On extrait 8 bits à partir du bit 3

3 8 sub

# Pose sur la pile BIN:00001110
```

Il est aussi possible d'effectuer des décalages de bits avec la fonction `shift`. Il faut lui indiquer de combien de bits on décale et dans quel sens (un nombre de bits négatif décale à gauche, sinon à droite) :

```
# On décale de 2 bits à gauche le nombre binaire BIN:00000001

BIN:00000001 -2 shift

# Pose sur la pile BIN:00000100

# On décale à droite d'un seul bit

1 shift

# Pose sur la pile BIN:00000010
```

La fonction `reverse` permet de "retourner" un nombre binaire :

```
# On retourne le nombre binaire BIN:11000111

BIN:11000111 reverse

# Pose sur la pile BIN:11100011
```

La fonction `not` permet d'appliquer un non binaire :

```
# On applique un non binaire à BIN:11000111

BIN:11000111 not

# Pose sur la pile BIN:00111000
```

Pour convertir un nombre binaire en nombre classique il faut utiliser la fonction `->num` :

```
# On récupère la valeur numérique de BIN:10011011

BIN:10011011 ->num

# Pose sur la pile 155
```

LA GESTION DU TEMPS

MOGWAI sait manipuler des informations concernant les dates et les durées.

Une date est un nombre qui représente le nombre d'intervalles de 100 nanosecondes qui se sont écoulés depuis minuit, le 1er janvier 0001.

Par exemple, la valeur représentant la date du 05/03/2012 à 16h45 est 6.3466562759E+17.

Bien sûr sous cette forme il n'est pas très pratique, c'est pourquoi MOGWAI possède toute une série de fonctions permettant d'effectuer des opérations sur les dates et les durées.

Récupérer la date courante

La fonction `now` retourne (pose sur la pile) la date courante de votre machine.

Récupérer les composants d'une date

Pour récupérer tous les composants (jour, mois, année, heure, etc..) d'une date il faut utiliser la fonction de conversion `->date` qui prend en paramètre une date (au format numérique) et retourne un record contenant tous les composants de cette date.

Les composants retournés sont les suivants (les clés du record retourné) :

day:	Jour du mois.
month:	Mois.
year:	Année.
hour:	Heure.
minute:	Minute.
second:	Seconde.
dayOfYear:	Numéro du jour dans l'année (ex 244ème jour).
dayOfWeek:	Numéro du jour dans la semaine (Dimanche=0, Lundi=1, etc).

Les composants retournés sont tous des nombres.

Opération	Résultat
now ->date	[day: 23 month: 5 year: 2025 hour: 12 minute: 19 second: 51 dayOfYear: 143 dayOfWeek: 5]

Cette méthode retourne tous les composants d'une date. Si vous n'avez besoin que d'un seul composant (par exemple l'année) il existe des fonctions ciblées pour ne retourner que le composant dont vous avez besoin :

->day	Retourne le jour d'une date.
->month	Retourne le mois d'une date.
->year	Retourne l'année d'une date.
->hour	Retourne l'heure d'une date.
->minute	Retourne les minutes d'une date.
->second	Retourne les secondes d'une date.

Créer une date de toutes pièces

Pour créer une date à partir de ses composants, il suffit de fournir un record contenant les composants de la date et d'utiliser à nouveau la fonction `->date` qui retournera cette date au format tick (nombre).

Il n'est pas nécessaire de fournir tous les composants, seuls le jour, le mois et l'année sont requis. Ceux qui sont omis sont considérés comme étant à zéro.

Opération	Résultat
[day: 15 month: 4 year: 2015] ->date	Crée la date 15/04/2015 à 00:00:00
[day: 15 month: 4 year: 2015 hour : 15] ->date	Crée la date 15/04/2015 à 15:00:00

Calculer des durées

Il est aussi possible de calculer des durées. Une durée étant une différence entre 2 dates, on peut assez facilement effectuer de tels calculs avec MOGWAI.

```
# On calcule le temps vraiment écoulé pendant une pause de 2450 ms

now -> 'begin'
2450 sleep
now -> 'end'
end begin - ->duration ?

# Resultat = [days: 0 hours: 0 minutes: 0 seconds: 2 milliseconds: 461]
# Soit 2 secondes et 461 millisecondes
```

Pour récupérer le temps écoulé entre 2 moments (2 dates) il suffit de soustraire la date d'arrivée à la date de départ et d'utiliser la fonction `->duration` pour extraire les composants de cette durée. La valeur retournée est un record composé de 5 clés :

days:	Nombre de jours de la durée.
hours:	Nombre d'heures de la durée.
minutes:	Nombre de minutes de la durée.
secondes:	Nombre de secondes de la durée.
milliseconds:	Nombre de millisecondes de la durée.

Il est aussi possible de récupérer ces composants directement. Dans ce cas on obtient le total de la durée dans l'unité demandée.

Dans notre exemple précédent, si on demande le total de secondes, avec la fonction `->seconds` on obtiendra la valeur de 2.4551168 secondes.

->days	Durée exprimée en jours.
->hours	Durée exprimée en heures.
->minutes	Durée exprimée en minutes.
->seconds	Durée exprimée en secondes.
->miliseconds	Durée exprimée en millisecondes.

```
# On calcule le temps écoulé pendant une pause de 2450 ms

now -> 'begin'
2450 sleep
now -> 'end'
end begin - ->seconds ?

# Resultat = 2.4551168 secondes
```

```
# On calcule le temps écoulé pendant une pause de 2450 ms
# Avec une écriture plus compacte.

now -> 'begin' 2450 sleep now begin - ->seconds ?
```

DECLARATION DE FONCTIONS

En plus de toutes les fonctions fournies en standard par MOGWAI, vous pouvez créer vos propres fonctions qui s'appellent des programmes (type `.program`).

Il existe différentes manières de déclarer des fonctions, nous allons les voir les unes après les autres. Les différences tiennent dans le niveau de sécurisation des paramètres passés (vérification des types des paramètres plus ou moins évoluée).

Une fonction doit être déclarée avant de pouvoir être utilisée.

Une fonction est délimitée par les symboles « » (ALT 174 et 175).

A partir de maintenant, les mots 'fonction' et 'programme' seront synonymes.

Déclaration d'une fonction de base

Une fonction de base prend tous ses paramètres sur la pile. Lors de sa déclaration rien n'est dit sur les paramètres attendus. Bien entendu, une fonction peut n'avoir aucun paramètre.

```
# On crée une fonction carre qui prend un nombre en paramètre et retourne son carré
# On prend le paramètre sur la pile, on le duplique puis on effectue leur multiplication
# Le résultat reste sur la pile, la fonction est terminée.

to 'carre' do « dup * »

# Pour l'utiliser :

5 carre

# Pose sur la pile la valeur 25 (le carré de 5)
```

Une fonction, en plus d'utiliser toutes celles fournies par MOGWAI, peut utiliser celles que vous définissez. Par exemple pour créer la fonction 'cube' qui va calculer le cube d'un nombre on va utiliser la fonction 'carre' que nous avons défini plus haut :

```
# On crée une fonction cube qui prend un nombre en paramètre et retourne son cube
# On prend le paramètre sur la pile, on le duplique puis on calcule son carre
# Puis on multiplie les 2 valeurs pour obtenir le cube.
# Le résultat reste sur la pile, la fonction est terminée.

to 'cube' do « dup carre * »

# Pour l'utiliser :

5 cube

# Pose sur la pile la valeur 125 (le cube de 5)
```

Déclarer une fonction avec les types des paramètres vérifiés

Il est possible de créer une fonction avec des paramètres vérifiés au moment de l'appel. Ceci évite d'avoir à effectuer toutes les vérifications dans le corps de la fonction. Ce sont des opérations qui peuvent être fastidieuses et couteuses en temps.

Par exemple dans la fonction précédente 'carre' rien n'est vérifié, si on passe une chaîne de caractères en paramètre au lieu d'un nombre une erreur sera levée au moment d'effectuer la multiplication. Idéalement il faudrait vérifier que le type du paramètre est bien `.number` avant de faire quoi que ce soit.

Pour éviter cela on peut dès la déclaration de la fonction indiquer les paramètres attendus et leur type :

```
# On crée une fonction carre avec vérification du type du paramètre d'entrée.

to 'carre' with [x: .number] do « x x * »

# Le paramètre attendu sera placé dans la variable locale 'x' et son type sera vérifié.
# Si le nombre de paramètre passé est insuffisant ou que le type d'un des paramètres est
# Erroné, une erreur est levée.

5 carre

# Posera sur la pile 25

"EEE" carre

# Le type est incorrect !
# Une erreur est levée avec un message d'explication :
# bad argument type
# ->safeVars
# .number expected but .string found for 'x' parameter

clear carre

# Si on vide la pile et qu'on appelle la fonction sans aucun paramètre
# Une erreur est levée :
# too few arguments
# ->safeVars
```

On peut définir si besoin un nombre illimité de paramètres vérifiés :

```
# On crée une fonction qui calcule un point d'une droite avec la formule
# y=a*x+b soit en RPN a x * b +

to 'fx' with [a: .number b: .number x: .number] do « a x * b + »

# Pour les valeurs y=5x+9 avec x=156 on appelle

5 9 156 fx

# Ce qui posera sur la pile 5*156+9 soit 789
```

Si on a besoin de passer un paramètre sans vérifier son type il faut utiliser le type `.any` au lieu d'un type précis :

```
# On crée une fonction qui affiche un message particulier si le type est un nombre.

to 'nPrint' with [x: .any] do
«
    if (x ->type .number ==) then
    {
        "C'est un nombre !" ?
    }
    else
    {
        "Ce n'est pas un nombre." ?
    }
»

# On appelle avec un nombre en paramètre…

234 nPrint

# Affichera le message "C'est un nombre !"

# Si on appelle avec un boolean…

true nPrint

# Affichera le message "Ce n'est pas un nombre."
```

Déclarer une fonction avec des paramètres nommés

Il est aussi possible de déclarer une fonction dont les paramètres sont explicitement nommés et les types vérifiés (ceinture et bretelles). On peut même définir des valeurs par défaut.

Pour la lisibilité du code c'est bien plus clair et la sécurité est maximale avec cette manière de faire. Les paramètres sont passés via un record dont les clés sont les noms des paramètres et les valeurs celles des paramètres.

Si on déclare notre fonction précédente 'fx' avec cette méthode ça donnerai ceci :

```
# On crée une fonction qui calcule un point d'une droite avec la formule
# y=a*x+b soit en RPN a x * b +

to 'fx' params [a: .number b: .number x: .number] do « a x * b + »

# Pour les valeurs y=5x+9 avec x=156 on appelle

[a: 5 b: 9 x: 156] fx

# Ce qui posera sur la pile 5*156+9 soit 789
```

Il est possible d'appeler ce type de fonction d'une manière moins RPN (paramètres puis fonction) en incluant le nom de la fonction en 1ère position du record des paramètres (cette notation n'est possible qu'avec l'appel de fonctions, ce type de notation pour un record n'existe pas ailleurs) :

```
# On crée une fonction qui calcule un point d'une droite avec la formule
# y=a*x+b soit en RPN a x * b +

to 'fx' params [a: .number b: .number x: .number] do « a x * b + »

# Pour les valeurs y=5x+9 avec x=156 on appelle

[fx a: 5 b: 9 x: 156]

# Ce qui posera sur la pile 5*156+9 soit 789
```

Pour déclarer des valeurs par défaut, il suffit de stipuler le type et la valeur par défaut dans une liste. Ainsi, si le paramètre n'est pas fourni, c'est la valeur par défaut qui sera utilisée.

```
# On crée une fonction 'foo' qui possède un paramètre booléen save: optionnel
# Si on ne le précise pas, il aura comme valeur par défaut true

to 'foo' params [id: .number name: .string save: (.boolean true)] do
«
    " " ?
    "id   = {! id}" eval ?
    "name = {! name}" eval ?
    "save = {! save}" eval ?
»

[foo id: 10 name: "Dupont Louis"]
[foo id: 30 name: "Martin Jean-Louis" save: false]

# Lors du 1er appel, le paramètre save: aura comme valeur true (valeur par défaut)
# Lors du 2ème appel, il aura la valeur false (paramètre fourni explicitement)
```

Récupérer la liste des fonctions déclarées

La fonction `funcs` retourne la liste des fonctions déclarées sous la forme d'une liste de noms. Il est par exemple possible, en cours de programme, de vérifier qu'une fonction existe bien avant de l'utiliser.

```
# On crée les fonctions carre et cube

to 'carre' do « dup * »

to 'cube' do « dup carre * »

# On liste les fonctions existantes

funcs

# Pose sur la pile la liste ('carre' 'cube')
```

GESTION DES ERREURS

En cas de problème MOGWAI, comme la plupart des langages de programmation, lève une erreur et stoppe le programme.

Il est possible de gérer le déclenchement d'une erreur et de faire en sorte que le programme ne plante pas bêtement.

L'instruction trap

Pour éviter d'arrêter le programme en cas d'erreur, l'instruction `trap` permet de "protéger" un bloc de code. Si une erreur se produit, le code protégé s'arrête et le code continue juste après l'instruction `trap`.

Attention, quand une erreur se produit, l'état de la pile est souvent difficile à connaître, c'est là une des difficultés.

```
# On va générer une erreur en utilisant une variable qui n'existe pas encore.
# Le code sera protégé par l'instruction trap.

trap
{
    "Début du trap." ?

    10 a *

    "Ce message ne sera jamais affiché." ?
}

"Sortie du trap." ?
"Le code continue…" ?
```

L'instruction guard

L'instruction `guard` est un peu plus évoluée que `trap`. Elle permet d'exécuter du code si une erreur se produit.

```
# On va générer une erreur en utilisant une variable qui n'existe pas encore.
# Le code sera protégé par l'instruction guard.

guard
{
    "Début du guard" ?

    10 a *

    "Ce message ne sera jamais affiché" ?
}
else
{
    "Une erreur s'est produite dans le guard !" ?
}

"Sortie du guard" ?
"Le code continue…" ?
```

Connaître la dernière erreur levée

Savoir qu'une erreur s'est produite sans pour autant tuer le code c'est bien mais savoir quelle erreur a été levée c'est mieux pour être en mesure de pouvoir réagir.

La fonction `error.last` retourne le code de la dernière erreur générée.

La fonction `error.reset` permet de remettre à zéro (pas d'erreur) le code de la dernière erreur. Donc il est de bon ton de remettre à zéro cette information une fois que vous avez terminé de gérer la dernière erreur car elle ne se remettra pas à zéro toute seule.

```
# On va générer une erreur en utilisant une variable qui n'existe pas encore.
# Le code sera protégé par l'instruction guard.

guard
{
    "Début du guard" ?

    10 a *

    "Ce message ne sera jamais affiché" ?
}
else
{
    "L'erreur " ?? error.last ?? " s'est produite !" ?
    error.reset
}

"Sortie du guard" ?
```

Lever une erreur artificiellement

Il est possible de lever une erreur en utilisant la fonction `error.throw` qui prend en paramètre le numéro de l'erreur à lever.

Liste des principales erreurs

Erreur	Libellé
0	aucune erreur
1	too few arguments
2	bad argument type
3	parse error
4	record definition error
6	unknown type
7	bad name
8	bad key
9	syntax error
10	unknown word error

Erreur	Libellé
11	null eval error
12	unknown var
13	empty stack
14	bad argument value
15	internal error
16	busy error
17	halt encounted
18	bad type error
19	engine error
20	file access error
21	not a number error
24	size too small
25	illegal name
40	name already exists
41	unknown name
200	var already exists
201	function already exist
202	unknown function
203	const already exists
204	const can't be deleted
205	const can't be modified

FAIRE UNE PAUSE

Il est parfois nécessaire de faire une pause dans un programme ou une fonction. MOGWAI propose 2 fonctions pour cela qui ont un fonctionnement légèrement différent.

La fonction sleep

Avec la fonction `sleep` le programme est complètement suspendu pendant le nombre de millisecondes passé en paramètre. Cela sous entant que pendant la pause, absolument plus ne se passe. Les événements ne sont pas reçus et traités, même chose pour les timers. Tout le moteur d'exécution est suspendu.

Si vous n'utilisez pas d'événements, ni de timer, cela n'a pas d'importante, sinon il faut plutôt utiliser la fonction `wait`.

La fonction wait

Avec la fonction `wait`, le programme est suspendu pendant le nombre de millisecondes passé en paramètre mais les événements et les timers continuent de fonctionner.

```
# On va afficher les nombres de 1 à 100
# Avec une pause de 250 millisecondes entre chaque

1 100 for 'i' do
{
    i ?
    250 wait

    # On aurait pu écrire aussi 250 sleep
}
```

SORTIR D'UNE FONCTION, D'UNE BOUCLE OU DU PROGRAMME

Le flux d'un programme peut être "cassé" par les 4 fonctions `mogwai.exit`, `mogwai.halt`, `break` et `return`.

La fonction mogwai.exit

Il est possible à tout moment d'arrêter le programme, d'en sortir. C'est la fonction `mogwai.exit` qui s'occupe de ça.

Quand un programme se termine sans erreur (arrêt normal ou provoqué par l'instruction `mogwai.exit`), la fonction réservée `MOGWAI.onStop` est automatiquement exécutée par MOGWAI. Si elle est définie dans votre code elle sera appelée automatiquement :

```
# On définit la fonction qui sera exécutée à la fin
# Normale du programme

to 'MOGWAI.onStop' do
«
    "Le programme vient de se terminer" ?
»

# On effectue une tâche infinie
# Mais si une valeur < 50 sort on arrête le programme

forever do
{
    # On tire un nombre au hasard et on le stocke
    # Dans la variable locale 'r'
    rand 1000 * ->int -> 'r'

    # On l'affiche et on fait une petite pause
    r ? 250 wait

    # Si le nombre est < à 50 alors on arrête le programme
    if (r 50 <) then {exit}
}

# Du coup le code qui suit ne sera jamais exécuté
# Mais la fonction MOGWAI.onStop sera automatiquement exécutée

"Code mort !" ?
```

La fonction mogwai.halt

La fonction `mogwai.halt` se comporte exactement comme la fonction `mogwai.exit`, mais elle lève une erreur 17, "halt encounted" au lieu de ne rien dire du tout. C'est donc un arrêt sur erreur.

Quand un programme se termine sur une erreur (`mogwai.halt` lève une erreur), la fonction réservée `MOGWAI.onError` est automatiquement exécutée par MOGWAI. Si elle est définie dans votre code elle sera appelée automatiquement :

```
# On définit la fonction qui sera exécutée
# Si une erreur est levée dans le programme

to 'MOGWAI.onError' do
«
    "Une erreur s'est produite !" ?
»

# On effectue une tâche infinie
# Mais si une valeur < 50 sort on arrête le programme
# Avec halt qui provoque un arrêt sur erreur

forever do
{
    # On tire un nombre au hasard et on le stocke
    # Dans la variable locale 'r'
    rand 1000 * ->int -> 'r'

    # On l'affiche et on fait une petite pause
    r ? 250 wait

    # Si le nombre est < à 50 alors on arrête le programme
    if (r 50 <) then {halt}
}

# Du coup le code qui suit ne sera jamais exécuté
# Mais la fonction MOGWAI.onError sera automatiquement exécutée

"Code mort !" ?
```

La fonction break

Quand vous êtes dans une boucle (voir le chapitre `BOUCLES`) il est possible d'en sortir "de force" avec la fonction `break` qui est utilisable dans les boucles `while`, `for`, `foreach`, `during`, `repeat` et `forever`.

```
# On affiche les nombres de 1 à 100 avec une boucle for
# Si le nombre en cours est > à 10 on sort de la boucle

1 100 for 'i' do
{
    i ?

    if (i 10 >) then {break}
}

# Le code continue ici

"Suite du programme…" ?
```

La fonction return

Elle permet de sortir prématurément d'une fonction.

```
to 'displayValue' with [value: .number] do
«
    # On affiche la valeur telle qu'elle
    # Sauf si c'est la valeur 5 qu'on remplace par un message

    if (value 5 !=) then
    {
        value ?
        return
    }

    "Valeur 5 interdite !" ?
»

1 10 for 'i' do
{
    i displayValue
}
```

CREATION AUTOMATIQUE DE VARIABLES

La fonction ->vars

La fonction `->vars` évite beaucoup d'opérations lorsqu'on souhaite créer des variables locales depuis une source telle qu'un record, ou depuis la pile.

->vars depuis un record

Si vous avez un record et que vous devez récupérer les valeurs incluses pour les manipuler, la solution basique consiste à récupérer les valeurs pour les affecter à la main à des variables locales avant de les traiter.

```
# On doit traiter les valeurs portées par les clés x: et y:
# D'un record qui est stocké dans la variable locale 'r'

[x: 50 y: 30] -> 'r'

# Méthode basique, on récupère à la main les valeurs
# Pour les stocker dans des variables locales ayant le même nom que les clés

r x: get -> 'x'
r y: get -> 'y'

# Maintenant, on peut traiter les valeurs
# Par l'intermédiaire des variables locales
```

On peut simplifier le code en utilisant la fonction `->vars`

```
# On doit traiter les valeurs portées par les clés x: et y:
# D'un record qui est stocké dans la variable locale 'r'

[x: 50 y: 30] -> 'r'

# Méthode plus rapide et automatique
# Pour les stocker dans des variables locales ayant le même nom que les clés

r ->vars

# Maintenant, on peut traiter les valeurs avec les variables locales x et y
# Qui ont été créées automatiquement par ->vars et qui portent les valeurs
# qu'ont les clés correspondantes dans le RECORD.
```

->vars depuis la pile

Il est possible de prélever automatiquement des éléments de la pile et de les stocker dans des variables locales avec `->vars`.

Il suffit de stipuler la liste des variables à créer en paramètre. Le nombre d'éléments correspondant au nombre de variables dans la liste seront prélevés de la pile et stockés dans les variables locales correspondantes. S'il n'y a pas assez d'éléments sur la pile pour remplir toutes les variables listées, la fonction lève une erreur sans modifier la pile.

```
# On pose sur la pile 5 éléments pour le test

56
"HELLO"
12.34
(1 2 3)
true

# On enregistre les éléments de la pile dans les variables locales a b et c

('a' 'b' 'c') ->vars

# Les variables a b et c ont été créées avec les valeurs
# a=12.34, b=(1 2 3) et c=true
# Les éléments "HELLO" et 56 n'ont pas été prélevé
```

La fonction ->safeVars

Avec la fonction `->safeVars` il est possible de vérifier que les valeurs présentes sur la pile sont bien celles attendues. Vous pouvez vérifier leur nombre et leur type, et affecter automatiquement des variables locales avec les valeurs de la pile. En cas de non-conformité une erreur est levée.

```
# On pose sur la pile 5 éléments pour le test

56
"HELLO"
12.34
(1 2 3)
true

# On enregistre les éléments de la pile dans les variables locales a b et c
# On détermine quels types sont attendus

[a: .number b: .list c: .boolean] ->safeVars

# Les variables a b et c ont été créées avec les valeurs
# a=12.34, b=(1 2 3) et c=true
# Les éléments "HELLO" et 56 n'ont pas été prélevé
# Au passage ->safeVars a vérifier que la valeur prélevée sur la pile pour la variable
# 'a' est bien du type .number, pour 'b' du type .list et pour 'c' de type .boolean.
```

Cette fonction est utilisée automatiquement quand vous déclarez une fonction avec le mot clé with :

```
to 'carre' with [x: .number] do « x x * »

5 carre

# Posera sur la pile 25
```

La fonction ->params

La fonction `->params` permet de passer des paramètres nommés (des clés/valeurs dans un record) et de vérifier que les paramètres attendus sont bien présents et que leur type correspond. Si tout est correct, les variables locales correspondantes aux paramètres attendus sont automatiquement créées avec les valeurs correspondantes.

Cette fonction prend 2 record en paramètres. Le 1er contient les valeurs à récupérer, le second décrit les paramètres attendus et leur type.

Par exemple pour récupérer 2 paramètres, nommés nom et age, nom étant une chaîne de caractères et age un nombre, on aura comme record de définition des paramètres :

`[nom: .string age: .number]`

Donc pour passer « STEPHANE » pour le nom et 55 pour l'âge on aura :

`[nom: "STEPHANE" age: 55] [nom: .string age: .number] ->params`

Comme tout correspond, MOGWAI va créer les variables locales 'nom' avec comme valeur « STEPHANE » et 'age' avec comme valeur 55.

```
# On passe en paramètre un nom de type chaine de caractères
# Et un age qui est un nombre

[nom: "STEPHANE" age: 55] [nom: .string age: .number] ->params

nom ?
age ?
```

Si on passe des valeurs ayant un mauvais type, une erreur est levée :

```
[nom: "STEPHANE" age: "TROP VIEUX"] [nom: .string age: .number] ->params

# age n'a pas le bon type
# Une erreur est levée
```

Si on passe plus de paramètres qu'attendus, ils seront simplement ignorés. Par contre si on ne passe pas tous les paramètres attendus, une erreur est levée.

Pour passer un paramètre ayant n'importe quel type, il faut utiliser le type .any

```
[nom: "STEPHANE" age: 55 libre: true] [nom: .string age: .number libre: .any] ->params

nom ?
age ?
libre ?

# Affichera :
# STEPHANE
# 55
# true
```

Cette fonction est utilisée automatiquement quand vous déclarez une fonction avec le mot clé params :

```
to 'fx' params [a: .number b: .number x: .number] do « a x * b + »

[a: 5 b: 9 x: 156] fx

# Ce qui posera sur la pile 5*156+9 soit 789
```

EVALUATION DES OBJETS

MOGWAI permet de placer dans certains objets des références directes à des variables, des fonctions et même du code exécutable.

Les objets pouvant supporter cette possibilité sont les `record`, les `list` et les `chaînes de caractères`.

Quand vous utilisez des références directes, elles ne seront pas remplacée automatiquement par leur valeur à l'instant où vous les utilisez.

Evaluation d'une liste

Si vous avez une variable 'A' ayant comme valeur 100, et que vous posez sur la pile la liste `(4 5 A 50)`, vous aurez sur la pile `(4 5 A 50)` et non `(4 5 100 50)`. Pour que la liste utilise la vraie valeur de A vous devez l'évaluer grâce à la fonction `eval`.

Donc si vous posez sur la pile `(4 5 A 50)` et que vous utilisez `eval` juste après, vous aurez au final `(4 5 100 50)` sur la pile.

Evaluation d'un record

La même chose est possible avec un record :

`[x: 10 y: 50 z: A] eval` donnera `[x: 10 y: 50 z: 100]`

Evaluation d'une chaîne de caractères

Pour les chaînes de caractères, vous devez utiliser la notation du bloc de code dans lequel vous faite juste apparaître le nom de la variable à remplacer.

Si vous devez inclure la valeur de A dans une chaîne de caractères, vous pouvez par exemple écrire :

`"La valeur de A est {! A}" eval` ce qui donnera `"La valeur de A est 100"`

Le ! doit impérativement être collé au { de début de bloc de code sinon la séquence ne sera pas reconnue.

Utilisation de code directement dans les objets

Il est possible d'utiliser du code dans les objets vus précédemment :

```
# On va afficher la table de multiplication du 7

0 9 for 'i' do
{
  "7 x {! i} = {! i 7 *}" eval ?
}

# Ce qui affichera
# 7 x 0 = 0
# 7 x 1 = 7
# 7 x 2 = 14
# ...
# 7 x 6 = 42
# 7 x 7 = 49
# 7 x 8 = 56
# 7 x 9 = 63
```

On peut faire la même chose avec une liste:

Avec A qui vaut 100,

`(A {! A 2 *} {! A 3 *}) eval` donnera `(100 200 300)`

Ou avec un record :

`[x: A y: {! A 2 *} z: {! A 3 *}] eval` donnera `[x: 100 y: 200 z: 300]`

Notation plus rapide pour l'évaluation

La fonction `eval` peut être remplacée dans les `list` et les `record` par le signe `!` en première position.

Si on reprend nos exemples précédents :

`(! A {! A 2 *} {! A 3 *})` donnera `(100 200 300)`

`[! x: A y: {! A 2 *} z: {! A 3 *}]` donnera `[x: 100 y: 200 z: 300]`

Il n'est plus nécessaire d'appeler la fonction `eval`, l'évaluation est réalisée directement avant de poser la valeur sur la pile.