Langage de programmation Neon

Documentation

v*1.2*.fr

Cette documentation s'applique à la version 4.5 de Neon.

Introduction

Neon est un langage de script généraliste nativement concurrent destiné entre autres à rendre possible la programmation concurrente sur des machines ne disposant pas de système d'exploitation multitâches. Le langage Neon permet la description de programmes de calcul séquentiels et concurrents, l'interpréteur Neon permet de les exécuter.

Neon est designé pour être facile à comprendre et facile à utiliser. Cette documentation n'a pas de vocation pédagogique mais sert à recenser de manière exhaustive l'entièreté des fonctionnalités de Neon. Ce document n'est pas non plus destiné à être lu dans l'ordre, mais à ranger les informations dans des catégories.

Par Neon, on désigne à la fois le langage dans ses spécificités et toutes les déclinaisons du logiciel permettant de l'exécuter.

Le logiciel

Plateformes classiques

Cette section décrit l'utilisation du logiciel Neon sur les plateformes classiques *(*Linux, Windows*)*

L'interpréteur Neon dispose de deux modes *:* un mode console, permettant d'entrer du code et des expressions, et un mode exécution permettant d'exécuter directement des fichiers.

Le mode exécution directe

L'extension de fichiers officiellement supportée pour les programmes Neon est l'extension `.ne`. Il est recommandé de nommer tous les programmes Neon avec cette extension. Pour lancer un programme avec le mode exécution, il suffit d'envoyer le nom du fichier en argument à l'interpréteur Neon. Le nom du fichier peut être suivi d'arguments à envoyer au programme Neon.

Concrètement, il faut ouvrir un terminal et taper la commande suivante *:* `neon fichier.ne arg1 arg2 ... argn`

Le mode console interactive

Pour lancer le mode console interactive, il faut lancer l'interpréteur Neon sans aucun argument.

Pour en savoir plus sur la console, voir la section Plus sur la console

La plateforme `TI_EZ80` *(*TI-*83* Premium CE / TI-*84* Plus CE*)*

Cette plateforme est particulière car le matériel et le système d'exploitation fournis sont très minimalistes. Ainsi certaines choses diffèrent entre cette plateforme et les plateformes classiques. Néanmoins cette plateforme bénéficie d'un support logiciel bien plus important que les autres puisqu'un IDE complet est directement intégré à Neon. Cet IDE dispose d'un éditeur, d'une console, d'un explorateur de programmes et de fonctions de gestion des programmes.

Plus sur la console

Vue globale

Lors de l'ouverture de la console, un message de bienvenue sera affiché. Ce message indique la version exacte de Neon, la date exacte de compilation, le site web officiel ainsi qu'une invitation sur le serveur Discord officiel de Neon.

Lorsque la console est prête à recevoir une expression à évaluer, le curseur est sur une nouvelle ligne débutée par deux chevrons `>>`. Si l'on entre une expression, celle-ci sera évaluée, et son résultat ainsi que son type seront affichés sur une nouvelle ligne.

La console permet d'entrer du code sur plusieurs lignes, mais il faut pour cela que l'interpréteur détecte que le code entré n'est pas encore terminé *(*parenthèse non fermée, mot-clé `do` sans `end` correspondant, *...)*

La console fournit toutes les fonctionnalités de base d'une console comme par exemple un historique, la possibilité d'éditer la ligne courante.

De plus sur certaines plateformes *(*comme Linux*)*, le texte tapé dans la console ainsi que les résultats d'expressions sont automatiquement coloré syntaxiquement. Cela améliore la lisibilité et permet de plus facilement détecter les erreurs de syntaxe.

Les erreurs

Lorsqu'une erreur est déclenchée, la console affiche, dans l'ordre*:*

- Le type d'erreur *(*l'objet `Exception` ayant déclenché l'erreur*)*
- Le fichier en cours d'exécution au moment du déclenchement de l'erreur
- Un message d'erreur détaillant les raisons de l'erreur
- Une référence exacte sur le message d'erreur

Cette référence prend la forme de trois nombres séparés par `#`*:* `ERRCODE#SOURCEID#LINENO`.

- `ERRCODE` correspond au numéro d'identifiant de l'erreur déclenchée. À chaque numéro d'identifiant d'erreur correspond un message d'erreur paramétrique. Par exemple le code d'erreur *15* correspond au message paramétrique `"Trying to index <> whereas it is not indexable."` que l'on peut par exemple déclencher en évaluant `print[42]`
- `SOURCEID` correspond à l'identifiant de fichier source de Neon dans lequel l'erreur a été déclenchée. L'identifiant de chaque fichier peut être trouvé à la première ligne de chaque fichier `.c` du code source dans la définition de `NEON_SOURCE_ID`. Le fichier `main.c` du code source contient également un table de correspondance des fichiers sources et de leur identifiant.
- `LINENO` correspond à la ligne du code source à laquelle a été déclenchée l'erreur. Ensemble, `SOURCEID` et `LINENO` permettent d'identifier très précisément la cause d'une erreur déclenchée par l'interpréteur Neon.

Interruption d'un programme en cours d'exécution

Lorsqu'un programme Neon est en cours d'exécution, il est toujours possible de l'interrompre. Pour cela il faut appuyer sur `CTRL+C` *(*plateformes classiques*)*, et sur la touche `[ON]` *(*TI-*83* Premium CE/TI *84* Plus CE*)*

Lorsque l'on déclenche l'interruption d'un programme Neon, un invite demande d'effectuer un choix entre deux options *:*

- Stop currently running program *(*arrêter le programme en cours d'exécution*)*
- Open interactive console on current environment *(*Ouvrir une console interactive dans l'environnement actuel*)*

La première option va définitivement arrêter le programme et afficher une erreur `KeyboardInterrupt`. La deuxième option va démarrer une console interactive dans laquelle il est possible d'exécuter du code, évaluer des expressions, modifier les variables, etc. Lors de la sortie de cette console, le programme mis en pause va continuer là où il s'était arrêté, mais en intégrant toutes les modifications effectuées dans la console.

Sommaire

Préambule sur la syntaxe et la sémantique

Lorsque l'on écrit un programme Neon, on écrit du texte. Une suite de caractères. On peut considérer que le programme est simplement cette suite de caractères, dans ce cas-là on parle de syntaxe.

En revanche si on s'intéresse à comment ces suites de caractères sont interprétées par Neon, on parle de sémantique.

Il est donc toujours important de distinguer ces deux points de vues. Parfois nous parlerons d'expression ; il s'agit de la suite de caractères que l'on écrit dans un programme. Une expression s'évalue en un objet pendant l'exécution, et on peut mettre en correspondance une expression avec l'objet vers lequel elle s'évalue. Par exemple, si on écrit la variable `a` dans un programme, elle s'évaluera en sa propre valeur. De même pour l'expression `123.456` qui s'évaluera en le flottant *123.456*.

Dans la suite, nous parlerons de variables, de noms de fonctions, de noms de champs de containers, etc. Ces noms sont désignés sous le terme identificateur. Les identificateurs doivent suivre des règles précises pour être correctement lus par Neon.

Un identificateur doit obligatoirement commencer soit par une lettre minuscule, soit par une lettre majuscule, soit par un `_`. Pour les caractères suivants il est également possible d'utiliser des chiffres, un apostrophe `'` et un tilde `~`.

Cependant il est recommandé de n'utiliser le caractère `~` que pour les objets de module.

Enfin il existe une liste de noms de variables interdits car déjà utilisés par les mots-clés de Neon.

Voici cette liste exhaustive *:*

```
or
do
if
in
EE
ei
es
tr
xor
and
not
try
for
end
del
NaN
then
elif
else
pass
True
None
expt
atmc
break
while
False
local
await
import
atomic
except
method
return
foreach
continue
function
Infinity
parallel
```

La syntaxe de Neon est sensible à la casse et insensible aux espaces et aux tabulations. Les retours à la ligne sont tolérés dans les expressions après des parenthèses ouvrantes, des crochets ouvrants et des virgules.

Partie *1 :* Les objets et les variables

Tous les objets utilisables dans les programmes Neon sont regroupés au sein d'une seule et unique structure C appelée `NeObj`.

Voici la liste de tous les types d'objets Neon existants *:*

```
Integer
Real
String
Bool
List
NoneType
Exception
Built-in function
Function
Method
Container
Promise
```

1.1 - Les variables, listes et containers

1.1.1 - Conserver des objets unitaires avec des variables

Une variable est une case mémoire gérée par Neon qui peut contenir un objet. La plupart des objets créés dans Neon n'ont pas d'existence persistante. Ils sont créés lors de l'évaluation d'une expression, utilisés comme argument d'une fonction ou d'un opérateur, puis supprimés.

Pour conserver un objet dans la durée, il est possible de le stocker à l'intérieur d'une variable. Une variable est désignée par un nom. Son nom doit suivre les règles des identificateurs. De nombreux opérateurs permettent de gérer les variables, modifier leur contenu, les supprimer, etc.

Si une variable n'existe pas et qu'une expression essaie d'accéder à sa valeur, une erreur sera déclenchée. Pour créer une variable, il suffit de lui assigner une valeur comme si elle existait déjà. Exemple *:* `maNouvelleVariable = 12`. Ceci suffit à créer une variable qui aura pour valeur *12*.

En plus des variables il existe d'autres moyens de conserver des valeurs dans la durée. Il existe des objets qui permettent de ranger un certain nombre d'autres objets, d'y accéder et de les modifier *:* les listes et les containers. Leurs cas d'utilisation sont différents.

1.1.2 - Un objet de stockage à grande échelle *:* les listes

Les listes sont des objets qui ont la capacité de stocker un nombre *(*presque*)* illimité d'autres objets, de manière ordonnée. On les utilise pour stocker une grande quantité d'informations, souvent de même type. Une liste est comme une suite de variables, où chaque élément est identifié par un indice. Chaque élément d'une liste *(*donc identifié par un certain indice*)* est une case mémoire au même titre qu'une variable. Ainsi, les opérateurs qui permettent d'interagir avec les variables permettent également et au même titre d'interagir avec les cases des listes.

1.1.3 - Regrouper des informations avec les containers

Les containers sont également des objets contenant d'autres objets. Chaque champ d'un container est également une case mémoire au même titre qu'une variable, et peut être modifié grâce aux mêmes opérateurs que ceux qui modifient le contenu des cases des listes et des variables. Les containers sont utilisés pour regrouper un certain nombre d'informations différentes concernant le même objet.

Bien que ces trois entités *(*variables, listes et containers*)* aient des points communs, il est important de bien comprendre la différence fondamentale entre variables et listes/containers.

Une variable est certes une case mémoire qui peut stocker un objet, mais contrairement aux listes et aux containers, une variable **n'est pas** un objet. Elle contient un objet. La seule manière d'accéder à une variable est par son nom. Ainsi une variable peut contenir une liste ou un container, ou tout autre objet, mais ne peut pas contenir une autre variable.

En revanche, une liste ou un container est un simple objet. Il n'a pas de nom à proprement parler, et peut être créé comme résultat d'une expression comme il peut simplement être supprimé dès que l'on n'en n'a plus besoin. Pour le conserver il faut le stocker aussi *(*dans une liste, un container ou une variable*)*.

1.2 - Les objets

1.2.1 - L'organisation en mémoire, compteur de références et garbage collector

Tous les objets sont des structures passées sur la pile de la taille d'un pointeur + un octet. Certains objets comme les nombres, les booléens, les constantes None, les exceptions et les promesses sont stockés uniquement dans cette structure. Pour les autres objets, la structure contient un pointeur vers une zone dans le tas qui contient réellement l'objet. Quand on copie ces objets, seulement la structure passée sur la pile est copiée, le bloc dans le tas est le même pour toutes les copies. Chacun de ces blocs contient un compteur qui compte le nombre de références du bloc, et le bloc est libéré quand ce compteur atteint zéro.

Certains objets *(*les containers et le listes*)* contiennent d'autres objets. Cette situation peut créer des objets cycliques rendant le compteur de références insuffisant.

Pour cela Neon dispose en plus d'un Garbage Collector. Le Garbage Collector maintient une liste de tous les containers et de toutes les listes créés. À certains endroits précis de l'interpréteur, un ramassage est déclenché, et Neon marque tous les objets accessibles depuis les variables encore vivantes. Les objets non marqués lors de cette phase sont supprimés définitivement.

1.2.2 - Le type `Integer`

Les objets de type `Integer` sont des nombres entiers. Leur taille dépend du système cible, ce sont des entiers *64* bits sur des OS *64* bits, et des entiers *24* bits sur ez*80*.

Les `Integer` peuvent être créés par des constantes dans le code source du programme, combinés entre avec des opérateurs pour en générer de nouveaux ou envoyés dans des fonctions built-in pour en créer de nouveaux. Voir la section sur les opérateurs et les fonction built-in pour plus d'informations.

Il est possible de définir des constantes `Integer` en écriture décimale, mais également en hexadécimal et en binaire. Pour définir un `Integer` à partir de son écriture hexadécimale, il faut précéder l'écriture hexadécimale du préfixe `0x`, et du préfixe `0b` pour le binaire.

Pour l'hexadécimal, les lettres peuvent être des lettres minuscules ou majuscules.

1.2.3 - Le type `Real`

Les objets de type `Real` sont des nombres flottants. Comme pour les nombre entiers, leur taille dépend du système cible. Ce sont des flottants à double précision lorsque l'OS le permet, sinon ce sont des float sur *32* bits *(*même pour ez*80)*.

Les `Real` peuvent être créés par des constantes dans le code source du programme, combinés entre avec des opérateurs pour en générer de nouveaux ou envoyés dans des fonctions built-in pour en créer de nouveaux. Voir la section sur les opérateurs et les fonction built-in pour plus d'informations.

Note : Les opérateurs de calcul sur les nombres ne préservent pas tous le type. En effet, alors que la multiplication, la soustraction et la somme de deux entiers reste un entier, la division de deux entiers renverra toujours un nombre décimal. Ce comportement des opérateurs est explicité plus en détails dans la section sur les opérateurs.

1.2.3 - Le type `Bool`

Les objets de type `Bool` ne peuvent valoir que `True` ou `False`. Ce sont les deux seules valeurs possibles. Ces valeurs peuvent être obtenues en les écrivant littéralement dans un programme, ou grâce aux opérateurs booléens, qui seront décrits dans la section Opérateurs.

Toute expression booléenne *(*résultat des opérateurs de comparaison, des opérateurs logiques*)* s'évalue en `True` ou `False`.

Les booléens permettent de représenter des valeurs de vérité, et donc d'instrumenter des branchements conditionnels, des boucles, etc.

1.2.4 - Le type `List`

Une liste est un objet permettant de stocker une suite finie d'objets quelconques. Le terme informatique exact pour décrire cet objet serait tableau. Le langage de programmation Neon ne fait pas de différenciation entre ces deux termes. Lorsque l'on parle de liste chaînée, on parle explicitement de liste chaînée. On peut indexer des listes grâce à la syntaxe suivante *:* `list[i]`. L'indexation des éléments d'une liste commence à zéro, au lieu de commencer naturellement à *1*. Ainsi, `list[i]` ne correspond pas à l'élément i mais à l'élément i*+1*. Cela peut paraître déroutant pour quelqu'un qui n'y est pas habitué, mais c'est en réalité cohérent avec le reste du langage. Ce choix a été fait pour rester dans les conventions de la grande majorité des langages de programmation. Historiquement, cette convention date des années *70* avec le langage C notamment, où l'indexation des tableaux exprime en réalité un **décalage par rapport au premier élément** et non l'accès à l'élément n°i.

Une liste peut contenir n'importe quel type d'objet, y compris elle-même.

Lorsque l'on copie une liste, ses éléments ne sont pas copiés. Ainsi toute modification apportée à une copie de liste apparaîtra sur toutes les autres copies et l'originale. Pour vraiment copier une liste, et la rendre indépendante de la liste originale, il faut utiliser la fonction `copy` pour faire une copie profonde.

On peut obtenir des listes de plusieurs manières, en récupérant le retour de certaines fonctions ou certains opérateurs, mais également en écrivant des listes littérales. Ce sont des expressions qui sont évaluées en une liste.

La syntaxe pour créer une liste littérale est `[element1, element2, ..., elementn]`. Les éléments peuvent être des expressions quelconques. La liste contiendra les objets résultants de l'évaluation des expressions *(*voir la section sur les expressions pour de plus amples explications*)*.

1.2.5 - Le type `String`

Les String sont des chaînes de caractères ASCII null-terminated. Elles peuvent être crées comme constantes dans le code source du programme ou combinées entre elles/obtenues grâce à d'autres fonctions. Pour créer une chaîne de caractère comme constante, il faut entourer le texte de guillemets `"` ou bien d'apostrophes `'`. Exemple *:*

```
mot = 'voiture'
```
ou bien
```
mot = "voiture"
```

L'intérêt de ces deux différentes syntaxes est que pour définir une chaîne contenant le caractère `"`, on peut utiliser les délimiteurs `'` et inversement.

On peut définir certains caractères spéciaux dans les chaînes de caractères à l'aide du caractère \|. Voici la liste complète *:*
`\a` ⟶ caractère d'appel
`\b` ⟶ retour arrière
`\f` ⟶ nouvelle page
`\r` ⟶ retour chariot
`\v` ⟶ tabulation verticale

`\t` ⟶ tabulation horizontale
`\n` ⟶ nouvelle ligne

On peut notamment obtenir leur taille grâce à la fonction `len`, et accéder au caractère n°i grâce à la syntaxe `string[i]`. Cette syntaxe est la même que pour indexer des listes.

1.2.6 - Le type `None`

Le type `None` désigne un seul et unique objet *:* la constante `None`. C'est un objet particulier *:* c'est le seul objet de ce type, et il est également égal à son propre type. Cette constante peut être obtenue en écrivant None dans un programme, ou récupérée comme retour de certaines fonctions et de certains opérateurs. Cette valeur est une valeur par défaut qui sert souvent à indiquer que quelque chose ne contient rien ou que quelque chose ne sert à rien, ou est vide.

1.2.7 - Le type `Exception`

Les exceptions sont des objets qui désignent des types d'erreur. Quand une erreur est déclenchée dans Neon, cette erreur correspond à une certaine exception. On peut se servir de ces exceptions pour attrapper certains types d'erreurs via `try .. except`. Plus exactement, quand une exception est levée dans un bloc `try`, on peut lancer l'exécution d'un certain code en fonction de l'exception levée. Il y a des exceptions built-in, et des exceptions qui peuvent être créées par la fonction `createException`.

Voici la liste des exceptions built-in *:*
`SyntaxError` *:* Est déclenchée par une erreur de syntaxe
`FileSystemError` *:* Est déclenchée lorsqu'une fonction de lecture/écriture des fichiers échoue
`UnmeasurableObject` *:* Déclenchée par la fonction `len` sur un objet non mesurable
`UndefinedVariable` *:* Lorsqu'on essaie d'utiliser un objet non défini
`IncorrectFunctionCall` *:* Lorsqu'un appel à une fonction échoue
`MemoryError` *:* Erreur lors de l'allocation de ressources. Ce type d'erreur est généralement grave
`NonIndexableObject` *:* Tentative d'indexer un objet non indexable. Seules les listes et les chaînes de caractères sont indexables
`IncorrectIndex` *:* Déclenchée lorsque l'index d'une chaine de caractères ou d'une liste n'est pas un entier positif
`OutOfRange` *:* Lorsqu'une indexation dépasse la taille d'un objet
`IncorrectType` *:* Une type n'est pas compatible avec une certaine opération
`DivisionByZero` *:* Division euclidienne par zéro
`UnknownError` *:* Déclenchée lorsqu'il n'y a pas assez d'informations sur l'erreur
`AssertionFailed` *:* Déclenchée lorsque la fonction `assert` échoue
`DefinitionError` *:* Déclenchée principalement lorsque la définition d'un container est incorrecte au regard des informations dont dispose l'interpréteur sur ce type de container
`KeyboardInterrupt` *:* Déclenchée par un Ctrl-C dans le terminal. Sur plateforme `TI_EZ80`, cette erreur est déclenchée en appuyant sur la touche `ON`
`NotImplemented` *:* Déclenchée lors de l'appel à une fonctionnalité non implementée. Peut se produire lors de l'importation de modules non disponibles sur certaines plateformes comme le module `Graphics` *(*voir la section sur les graphiques*)*.
`ExitSignal` *:* Exception déclenchée par l'appel à `exit`.
`InternalError` *:* Exception déclenchée lorsque Neon rencontre une erreur inattendue. Lorsqu'une telle exception est déclenchée, il s'agit toujours d'un bug dans l'interpréteur Neon lui-même, jamais d'une erreur dans le programme Neon. Lorsqu'une exception `InternalError` est déclenchée, il faut signaler cela comme un bug. Une exécution normale ne doit jamais déclencher d'`InternalError`.
`DeserializationError` *:* Exception déclenchée lorsque la désérialisation d'un objet échoue. Un déclenchement de cette exception signifie toujours que le fichier de sauvegarde de l'objet est corrompu.
`Error` *:* Exception de base, utile pour déclencher des exceptions généralistes avec `raise`.

1.2.8 - Le type `BuiltInFunction`

Ce type regroupe toutes les fonctions présentes au chargement de l'interpréteur en mémoire.

Les fonctions sont des objets comme les autres, elle peuvent passer de variable en variable, être stockées dans des listes, etc. Tout objet contenant une fonction peut être exécuté en tant que fonction.

Pour exécuter une fonction, il suffit de suivre l'expression contenant la fonction par `(argument1, argument2, argument3...)` avec tous les arguments que l'on veut envoyer à la fonction.

Exemple*:*
Si la variable `maFonction` contient une fonction et que l'on veut exécuter cette fonction sur les arguments `a` et `b`, il suffit d'écrire l'expression *:* `maFonction(a, b)`. Cette expression s'évaluera en le résultat de la fonction sur les objets dénotés par `a` et `b`.

En plus de cette syntaxe pour appeler des fonctions, il existe une autre syntaxe, orientée objet. Au lieu d'écrire `maFonction(arg1, arg2, arg3...)`, on peut écrire `arg1.maFonction(arg2, arg3...)`. Ces deux syntaxes sont équivalentes.

Toutes les fonctions au sens général partagent la même syntaxe d'appel. Ainsi, les syntaxes présentées ici pour les fonctions de type `Built-in function` fonctionnent également pour les fonctions utilisateur et les méthodes.

Toutes les informations sur une fonction built-in peuvent être obtenues en tapant `help(fonction)` dans le terminal.

De manière générale, quand une fonction ne renvoie rien, elle renvoie en réalité None.

Voici dans l'ordre alphabétique la liste de toutes les fonctions built-in *:*

`abs(Real)` → `Real`*:*
Renvoie la valeur absolue d'un nombre.

`append(List, Any)` → `None`*:*
Cette fonction prend en argument une liste et un objet quelconque, et ajoute l'objet à la fin de la liste.

`assert(Bool)` → `None`*:*
Cette fonction prend en argument un booléen, et lève l'exception AssertionFailed si le booléen vaut False.

`bin(Integer)` → `String`*:*
Cette fonction prend un entier en argument et renvoie une chaîne de caractères correspondant à sa description en binaire. La chaîne de caractères n'est pas précédée du préfixe `'0b'` comme c'est notamment le cas en Python.

`ceil(Real)` → `Real`*:*
Renvoie l'arrondi par valeur supérieure d'un nombre.

`chr(String)` → `Integer`*:*
Cette fonction prend en argument un code ASCII *(*nombre entier*)* et renvoie le caractère correspondant.

`clear()` → `None`*:*
Cette fonction efface le terminal.

`copy(Any)` → `Any`*:*
Cette fonction renvoie la copie profonde d'un objet, en conservant les dépendances de pointeurs au sein de l'objet.

`cos(Real)` → `Real`*:*
Calcule le cosinus d'un angle en radians.

`count(List | String)` → `Integer`*:*
Cette fonction compte le nombre d'apparitions d'une sous-chaîne dans une chaîne de caractères ou compte le nombre d'objets présents dans une liste. Il faut indiquer d'abord la liste ou la chaîne de caractères, puis la sous-chaîne ou le sous objet.

`createException(String)` → `Exception`*:*
Cette fonction sert à créer des exceptions. Elle prend en argument une chaîne de caractères, et crée une exception ayant le nom donné en argument. Un nouveau mot-clé est créé avec le nom de cette exception, et elle est accessible directement avec ce mot-clé. La fonction `createException` renvoie également l'exception créée.

`deg(Real)` → `Real`*:*
Convertit en degrés un angle en radians.

`detectFiles(String)` → `List`*:*
Cette fonction renvoie une liste de tous les noms de fichiers du répertoire courant commençant par une certaine chaîne de caractères. Elle prend en argument la chaîne de caractères à comparer avec le début de chaque fichier.

`eval(String)` → `Any`*:*
Cette fonction prend en argument une chaîne de caractères correspondant à une expression Neon, et renvoie le résultat de l'évaluation de l'expression.

`exit()` → `None`*:*
Cette fonction n'attend aucun argument, et quitte l'interpréteur. Elle renvoie None

`exp(Real)` → `Real`*:*
Calcule l'exponentielle d'un nombre

`failwith(String)` → `None`*:*
Cette fonction prend en argument une chaîne de caractères, et quitte l'interpréteur en affichant cette chaîne de caractères.

`floor(Real)` → `Real`*:*
Renvoie l'arrondi par valeur inférieure d'un nombre.

`format(String, ...)` → `String`
Cette fonction prend en argument une chaîne de caractères ainsi qu'un nombre illimité d'objets. Elle renvoie une nouvelle chaîne de caractères, ayant remplacé tous les `<>` par la représentation de chaque objet donné en argument par la suite.

Exemple*:* `format("<> + <> = <>", 1,2,3)` renvoie `"1 + 2 = 3"`

`gc()` → `None`*:*
Cette fonction appelle le Garbage Collector.

`hash(Any)` → `Integer`*:*
Cette fonction prend un objet quelconque en argument et en calcule un hash, c'est-à-dire un entier destiné à représenter l'identité de l'objet. Calculer le hash du même objet renvoie toujours le même résultat.

This function takes any object as argument and computes a hash of this object, i.e an integer aimed to represent the identity of the object. Computing the hash of the same object will always give the same result, but computing the hash of any other object will at a very high probability be always different. In other words, the `hash` function can be supposed to be injective.

La représentation interne des entiers étant différente d'une plateforme à l'autre, le calcul du hash d'un objet peut renvoyer des valeurs différentes suivant la plateforme sur laquelle on effectue le calcul.

`help(Any)` → `None`*:*
Cette fonction affiche de l'aide liée à certains objets ou certains types d'objets. Voici les arguments possibles à la fonction `help` *:*
`help("modules")` ⟶ affiche tous les noms de modules présents dans la mémoire
`help("variables")` ⟶ affiche toutes les variables définies présentes dans la mémoire ainsi que leur type
`help("MonModule")` ⟶ affiche tous les objets liés au module MonModule présents dans la mémoire
`help(mon_objet)` ⟶ affiche mon_objet et son type

Dans le cas d'une fonction built-in, la fonction `help` affiche aussi le type des arguments attendus, le type de retour et une chaîne de caractères expliquant comment utiliser la fonction. Dans le cas d'une fonction utilisateur (ou d'une méthode utilisateur), affiche le nom de la fonction, les arguments attendus, et si une chaîne de caractères d'aide a été assignée par la fonction `setFunctionDoc`, affiche cette chaîne.

`hex(Integer)` → `String`:
Cette fonction prend un entier en argument et renvoie une chaîne de caractères correspondant à sa description en hexadécimal. Comme pour la fonction `bin`, la chaîne de caractères n'est pas précédée du préfixe `'0x'`.

`index(List, Any)` → `Integer`:
Cette fonction prend en argument une liste ou une chaîne de caractères.

Dans le cas où le premier argument est une liste, le second argument doit être un élément de cette liste. La fonction renvoie l'indice de la première apparition de l'objet dans la liste.

Dans le cas où le premier argument donné à la fonction est une chaîne de caractères, le second argument doit être une sous-chaîne de la première chaîne. La fonction renvoie l'indice de début de la première apparition de la sous-chaine dans la chaine de caractères.

`int(Any)` → `Integer`:
Cette fonction prend en argument un objet et le convertit en entier.

`input(...)` → `String`:
Cette fonction prend en argument une chaîne de caractères, l'affiche et attend du texte de l'utilisateur. Elle renvoie une chaîne de caractères correspondant au texte entré.

`insert(List, Any, Integer)` → `None`:
Cette fonction prend en argument une liste, un objet et un indice dans cette liste, et insère l'objet à l'indice indiqué.

`len(String | List)` → `Integer`:
Cette fonction prend une liste ou une chaîne de caractères et renvoie sa longueur.

`list(String)` → `List`:
Cette fonction transforme une chaîne de caractères en liste dont les éléments sont les caractères de la chaîne.

`listComp(String, Integer, Integer, Integer, String, String)` → `List`:
Cette fonction sert à fabriquer des listes de manière efficace. Elle prend en argument, dans l'ordre :

- Le nom d'une variable (dans une chaîne de caractères)
- Un indice de début
- Un indice de fin
- Un pas
- Une chaîne de caractères correspondant à une expression booléenne
- Une chaîne de caractères correspondant à une expression quelconque de Neon.

L'appel de `listComp("variable", debut, fin, pas, "condition", "expression")` renvoie une liste créée de cette manière :

```
l = []
for (variable, debut, fin, pas) do
    if (condition) then
        l.append(expression)
    end
end
```

`ln(Real)` → `Real`:
Calcule le logarithme népérien d'un nombre.

`loadNamespace(String)` → `None`:
Cette fonction charge dans la mémoire une copie des objets du module dont le nom est donné en argument sans préfixe.

`loadObj(String)` → `Any`:
Cette fonction permet de charger un objet sauvegardé à l'aide de `saveObj`. Elle prend en argument le nom du fichier de sauvagarde. Sur les systèmes de fichiers prenant en charge les extensions de fichiers, l'extension `.neobj` est automatiquement ajoutée au nom donné en argument pour rechercher le fichier à ouvrir.

La fonction `loadObj` ne peut pas charger un objet sauvegardé depuis une autre plateforme.

`log(Real)` → `Real`:* Calcule le logarithme base *10* d'un nombre.

`log2(Real)` → `Real`:
Calcule le logarithme base *2* d'un nombre

`nbr(String)` → `Real | Integer`:
Cette fonction prend en argument une chaîne de caractères représentant un nombre et la convertit en entier ou en décimal.

`ord(String)` → `Integer`:
Cette fonction prend en argument un caractère et renvoie son code ASCII.

`output(...)` → `None`:
Cette fonction prend en argument un nombre illimité d'objets de tous types, et affiche tous ces objets à la suite sans retour à la ligne.

`print(...)` → `None`:
Cette fonction attend un nombre indéfini de paramètres, et affiche la représentation en objet de chaque paramètre, séparé par un espace. Cette fonction affiche également un retour à la ligne. Elle renvoie None.

`rad(Real)` → `Real`:
Convertit en radians un angle en degrés.

`raise(Exception, String, ...)` → `None`:
Cette fonction prend en argument une exception, une chaîne de caractères et un nombre illimité d'arguments. Lorsqu'elle est appelée, elle lève l'exception donnée en argument en affichant la chaîne de caractères comme message d'erreur. La chaîne de caractères peut contenir les symboles `<>`, auquel cas ils seront remplacés par la représentation des objets supplémentaires donnés en arguments. Sur les plateformes supportant la coloration syntaxique, les objets donnés par `<>` seront syntaxiquement colorés. Si la fonction `raise` est appelée dans un bloc `try`, elle peut être interceptée et ne pas déclencher de réelle erreur. Voir la section sur la gestion d'erreurs pour plus de détails.

`randint(Integer, Integer)` → `Integer`:
Cette fonction prend en argument deux entiers *:* une borne inférieure et une borne supérieure, et renvoie un entier aléatoire compris entre ces deux bornes, borne supérieure exclue.

`readFile(String)` → `String`:
Prend en argument le nom d'un fichier texte et renvoie son contenu. Sur la plateforme TI_EZ*80*, les fichiers ne peuvent être que des AppVars.

`remove(List, Integer)` → `None`:
Cette fonction prend en argument une liste et un indice de cette liste, et supprime l'élément présent à cet indice.

`replace(String, String, String)` → `String`:
Cette fonction prend en argument trois chaînes de caractères, et remplace toutes les occurences de la deuxième chaîne dans la première chaîne par la troisième chaîne.

`reverse(String | List)` → `String | List`:
Cette fonction prend en argument une chaîne de caractères ou une liste et renvoie l'objet inversé sans modifier l'objet original.

`round(Real, Integer)` → `Real`:
Calcule l'arrondi d'un nombre à la précision demandée en deuxième argument.

`safeExec(String, List)` → `None`:
Cette fonction prend en argument un nom de fichier source Neon ainsi qu'une liste d'objets, et lance le programme dans un environnement indépendant en lui donnant la liste d'objets comme arguments.

`saveObj(String, Any)` → `None`:
Cette fonction permet de sauvegarder n'importe quel objet Neon *(*listes, fonctions, chaînes de caractères, containers, ...*)* dans un fichier. Ce procédé est souvent appelé sérialisation. Cet objet peut ensuite être rechargé dans n'importe quel environnement Neon et sera restauré à l'identique *(*voir la fonction `loadObj`*)*. Cette fonction prend en argument un nom de fichier *(*aucune extension n'est nécessaire, Neon ajoute automatiquement l'extension `.neobj` sur les systèmes de fichiers prenant en charge les extensions*)*, et un objet Neon.

`setAtomicTime(Integer)` → `None`:
Cette fonction change la période de changement de processus avec l'entier donné en argument.

`setColor(Const)` → `None`:
Cette fonction change la couleur du texte affiché dans le terminal après son appel. Les couleurs disponibles sont *:* `Blue`, `Red`, `Green`, `Purple`, `Orange`, `Grey`, `Default`.

Notez que les arguments à la fonction `setColor` sont des constantes symboliques. Ces constantes sont prédéfinies au chargement de l'interpréteur Neon.

La couleur `Default` est soit le blanc en mode blanc sur fond noir, soit le noir en mode noir sur fond blanc.

Sur les terminaux où c'est disponible, le rouge, le bleu et le violet sont affichés en gras.

`setFunctionDoc(Function | Method)` → `None`:
Cette fonction prend en argument une fonction utilisateur, et une chaîne de caractères, et définit cette chaîne de caractères comme message d'aide pour cette fonction. Ce message d'aide est affiché lorsque la fonction help est appelée avec cette fonction.

`sin(Real)` → `Real`:
Calcule le sinus d'un angle en radians.

`sortAsc(List)` → `None`:
Cette fonction trie une liste dans l'ordre croissant suivant l'ordre lexicographique ou l'ordre sur les nombres.

`sortDesc(List)` → `None`:
Pareil mais dans l'autre sens.

`sqrt(Real)` → `Real`:
Calcule la racine carrée d'un nombre.

`str(Any)` → `String`:
Cette fonction prend un objet quelconque et renvoie sa représentation textuelle évaluable par Neon.

`sub(String, Integer, Integer)` → `String`:
Cette fonction attend trois arguments *:* une chaîne de caractères et deux entiers. `sub(string, i1, i2)` renvoie la sous-chaîne de string commençant au caractère numéro *i1* jusqu'au caractère numéro *i2* exclu.

`tan(Real)` → `Real`:
Calcule la tangente d'un angle en radians.

`time()` → `Integer`:
Cette fonction renvoie le nombre de secondes écoulées depuis une certaine date dépendant du système sur lequel s'exécute le programme.

`type(Any)` → `String`:
Cette fonction prend en argument un objet et renvoie son type. Les types d'objet sont représentés par des constantes symboliques. Voici les types renvoyés par la fonction `type` :
`Bool` ⟶ booléen
`String` ⟶ chaîne de caractères
`Integer` ⟶ entier
`Real` ⟶ nombre décimal
`BuiltInFunction` ⟶ fonction built-in
`List` ⟶ liste
`Function` ⟶ fonction utilisateur
`Method` ⟶ méthode utilisateur
`Exception` ⟶ exception
`Promise` ⟶ promesse

La fonction type ne renvoie jamais de type `Container` car elle renvoie directement le nom du container.

`writeFile(String, String)` → `None`:
Cette fonction prend en argument un nom de fichier et une chaîne de caractères, et écrit cette chaîne de caractères dans le fichier dont le nom a été indiqué en argument. Si le fichier existe, son contenu est remplacé. Sur la plateforme TI_EZ*80*, les fichiers ne peuvent être que des AppVars.

1.2.9 - Le type `Function`

En Neon il est possible de créer des fonctions qui prennent des arguments en entrée et qui exécutent du code en fonction de ces arguments. Lorsqu'une fonction est définie, une variable avec le nom de la fonction est créée, avec pour valeur l'objet fonction correspondant. Tout objet fonction est appelable, comme c'est le cas pour les fonctions built-in. On peut appeler la fonction help dessus, et lui ajouter de l'aide grâce à la fonction setFunctionDoc.

La manière de créer des fonctions sera détaillée dans la section consacrée.

1.2.10 - Le type `Method`

Une méthode est également un objet défini par utilisateur, quasiment comme une fonction. La seule différence est que contrairement à une fonction où tous les arguments sont des variables locales à la fonction et où leur modification n'a aucun impact sur les variables à l'extérieur, le premier argument d'une méthode peut être modifié directement par la méthode. Concrètement, une fois qu'une méthode a fini d'être exécutée, l'interpréteur Neon récupère la valeur de la première variable locale correspondant au premier argument, et l'assigne à l'objet qui a été envoyé en premier argument de la méthode. Ainsi toute modification effectuée au premier argument à l'intérieur d'une méthode sera appliquée à l'objet qui aura été envoyé en premier argument.

La manière de créer des méthodes sera détaillée dans la section consacrée.

1.2.11 - Le type `Container`

Les containers sont des objets qui permettent de ranger de manière propre d'autres objets, en leur donnant chacun un nom à l'intérieur du container. Chaque container possède également un nom qui le caractérise parmi les autres containers. Tous les containers définis avec le même nom doivent posséder exactement les mêmes champs. Par souci de clarté, les noms de containers doivent commencer par une majuscule. À part les noms de module, ce sont les seuls objets à devoir commencer par une majuscule.

Exemple *:* Définition d'un container Personne

```
monContainer = Personne(prenom: "Paul", nom: "Durand", age: 34, enfants:["Pierre",
"Jacques"])
```

Ce container est de type `Personne`, et contient les champs `prenom`, `nom`, `age` et `enfants`. La première apparition d'un container d'un certain type dans le code fixe les noms des champs pour ce type de container. C'est-à-dire qu'à partir du moment où le premier objet de type `Personne` aura été défini comme ci-dessus, tous les containers de type `Personne` devront contenir les champs `prenom`, `nom`, `age` et `enfants`. Il n'est pas nécessaire de définir les champs dans l'ordre. Tant que tous les champs sont présents, la définition est correcte.

Comme pour les listes, la copie d'un container n'entraîne pas la copie des objets qu'il contient. Ainsi un container peut se contenir lui-même, et une modification de champ dans un container entraînera la modification de ce champ dans toutes les copies qui ont été faites de cet objet et l'original. Pour vraiment copier un container il faut utiliser la fonction `copy`.

Pour accéder à un champ d'un container, il faut utiliser l'opérateur `>>`. Si on reprend l'exemple du container défini plus haut, `monContainer>>prenom` correspond à la variable qui contient `"Paul"`.

On peut utiliser cette syntaxe à la fois pour accéder aux noms des champs *:* print*(*monContainer>>prenom*)* et à la fois pour modifier les champs *:* `monContainer>>prenom = "Martine"`

1.2.12 - Le type `Promise`

Les objets de type promise ne peuvent être obtenus que par retour du lancement en parallèle d'un processus. Neon permet le lancement de fonction en parallèle grâce au mot-clé `parallel`.

Quand on lance une fonction en parallèle avec `parallel fonction(arg1, arg2, args...)`, une promesse est renvoyée. Cette promesse est un objet qui ne sert à rien pendant que le processus tourne, si ce n'est identifier le processus qui l'a renvoyée, car elle est unique pour chaque processus. Tant que le processus n'a pas terminé, son type est `"Promise"`. Une fois que la fonction lancée en parallèle a terminé et renvoyé sa valeur, la promesse que le mot-clé `parallel` avait renvoyée *(*et toutes ses copies*)* vont automatiquement se transformer en la valeur de retour de la fonction. Si la fonction ne renvoie rien, c'est qu'elle renvoie en réalité `None`, et la promesse prendra la valeur de `None`. Seules les fonctions utilisateur peuvent être exécutées en parallèle *(*pas les méthodes, ni les fonctions built-in*)*. Les fonctions built-in sont exécutées de manière atomique, et la fonction `setAtomicTime` permet de décider de la fréquence de passage d'un processus à l'autre.

La manière de gérer les processus sera détaillée dans la section dédiée.

1.2.13 - Le type `Const`

Le type `Const` correspond à des constantes symboliques. Les objets de ce type sont des valeurs uniquement définies par leur nom. Les valeurs de type `Const` sont par exemple utilisées pour les types des objets et les couleurs. Les types des objets et les couleurs sont des objets de type `Const` pré-définis dans l'interpréteur Neon.

Pour créer des constantes symboliques, il faut utiliser l'instruction `define`. Cette instruction prend en argument une liste d'un nombre quelconque d'identifiants, et définit pour chacun de ces identifiants une variable portant le même nom. Chaque variable a pour valeur la constante symbolique du même nom.

Exemple *:*

```
define(Vrai, Faux, JeSaisPas) # Ces mots sont désormais utilisables comme des
constantes symboliques
```

Partie *2 :* Les expressions

Les expressions sont à la base de tout code Neon. Une expression est une construction syntaxique qui peut être évaluée en un objet. Toute expression est évaluée en l'un des objets détaillés dans la **Partie I.** Une expression sert à

décrire un calcul. Les constantes littérales sont les expressions les plus basiques. Elles sont évaluées directement en leur objet associé. Une variable (désignée par son nom) est également une expression, et est évaluée en la valeur de la variable.

Des expressions plus compliquées peuvent être créées en combinant d'autres expressions à l'aide d'opérateurs et de fonctions.

Ces opérateurs et fonctions attendent des arguments (ou opérandes), qui, au moment de l'évaluation (ou de l'exécution) de la fonction ou de l'opérateur doivent être des objets. Pour envoyer un objet en tant qu'argument à une fonction ou un opérateur, il suffit d'écrire à la place de l'argument une expression qui s'évalue en l'objet voulu.

2.1 - Les opérateurs

Voici la liste de tous les opérateurs, et leur effet sur les objets de différents types :

2.1.1 - Opérateurs binaires

`+` :
Cet opérateur additionne deux nombres, concatène deux chaînes de caractères et concatène deux listes.

`*` :
Cet opérateur effectue une multiplication entre deux nombres. Le produit entre un entier `n` et une liste va renvoyer la concaténation de cette liste avec elle-même `n` fois. Même chose pour le produit entre une chaîne de caractères et un entier.
Le produit entre une liste (ou une chaîne de caractères) et un booléen a le même comportement que si le booléen était interprété comme *1* ou *0*.

`-` :
Cet opérateur effectue une soustraction entre deux nombres.
La soustraction entre une chaîne de caractères et un entier `n` retire les `n` derniers caractères de la chaîne.
La soustraction entre deux chaînes de caractères renvoie la première chaîne moins toutes les occurrences de la deuxième.

`/` :
Bien que cela puisse paraître étonnant, cet opérateur effectue une simple division entre deux nombres, rien d'autre. Il est vrai qu'on aurait pu imaginer des utilisations de l'opérateur `/` pour toutes sortes de types, mais il faut parfois être raisonnable.

`**` :
Opérateur exposant. `a ** b` renvoie `a` à la puissance `b`.

`==` :
Opérateur de comparaison entre objets. Il renvoie True si et seulement si les deux objets sont égaux.

`!=` :
Renvoie True si les deux objets sont différents.

`>=` :
Comparaison entre deux nombres. Renvoie True si et seulement si l'opérande de gauche est supérieure ou égale à l'opérande de droite.

`<=` :
Comparaison entre deux nombres. Renvoie True si et seulement si l'opérande de gauche est inférieure ou égale à l'opérande de droite.

`<` :
Comparaison entre deux nombres. Renvoie True si et seulement si l'opérande de gauche est inférieure strictement à l'opérande de droite.

`>` *:*
Comparaison entre deux nombres. Renvoie True si et seulement si l'opérande de gauche est supérieure strictement à l'opérande de droite.

`and` *:*
Effectue un ET logique entre deux booléens. Cet opérateur est paresseux *:* si l'opérande de gauche s'évalue à `False`, l'opérande de droite n'est pas évalué et l'opérateur renvoie `False`.

`or` *:*
Effectue un OU logique entre deux booléens. Cet opérateur est paresseux *:* si l'opérande de gauche s'évalue à `True`, l'opérande de droite n'est pas évalué et l'opérateur renvoie `True`.

`xor` *:*
Effectue un OU EXCLUSIF logique entre deux booléens.

`=>` *:*
Renvoie le résultat de l'implication logique entre deux booléens.

`=` *:*
Opérateur d'affectation. L'opérande de gauche doit être soit une variable, soit un index de liste soit un attribut de container, l'opérande de droite peut être n'importe quel objet. Cet opérateur a pour effet de changer la valeur contenue dans l'opérande de gauche. Une affectation renvoie None.

Si l'opérande de gauche est une variable qui existe, son contenu sera simplement modifié. Sinon, si la variable n'existe pas encore, elle sera préalablement créée.

`->` *:*
Cet opérateur est également un opérateur d'affectation. L'objet se place à gauche, et la variable/index de liste/attribut de container se place à droite. Contrairement à l'opérateur `=`, cet opérateur renvoie également une copie de la valeur assignée. Cela permet d'effectuer des affectations en chaîne *:* `6 -> a -> b -> c`

`+=` *:*
Opérateur binaire *;* `a += b` correspond exactement à `a = a + b`.

`-=` *:*
Opérateur binaire *;* `a += b` correspond exactement à `a = a - b`.

`/=` *:*
Opérateur binaire *;* `a += b` correspond exactement à `a = a / b`.

`*=` *:*
Opérateur binaire *;* `a += b` correspond exactement à `a = a * b`.

`%` *:*
Cet opérateur est un opérateur binaire qui attend deux entiers. Il renvoie le reste de la division euclidienne entre l'opérande de gauche et l'opérande de droite.

`//` *:*
Cet opérateur est un opérateur binaire qui attend deux entiers. Il renvoie le reste de la division euclidienne entre l'opérande de gauche et l'opérande de droite.

Bien que l'opérateur de division soit un simple opérateur entre nombres, j'ai quand même craqué pour cet opérateur. Lorsque l'une des opérandes est un entier `a` et que l'autre opérande est une chaîne de caractères de taille `n`, l'opérateur renvoie une nouvelle chaîne composée des `n//a` premiers caractères de la chaîne originale.

`<-` *:*
Cet opérateur attend une chaîne de caractères à gauche et un objet quelconque à droite. L'opérateur crée une variable avec le nom indiqué à gauche, même si le nom n'est pas un nom d'identificateur valide, et lui affecte l'objet indiqué à

droite. Si la variable existe déjà, l'opérateur change simplement sa valeur avec le nouvel objet. L'opérateur renvoie également une copie de l'objet.

`EE` *:*
Cet opérateur correspond à la mise à la puissance de dix. `a EE b` est égal à `a * 10 ** b`.

`in` *:*
Cet opérateur attend un objet quelconque à gauche et une liste à droite, et renvoie True si et seulement si la liste contient l'opérande de gauche.

`<->` *:*
Cet opérateur attend à gauche et à droite des variables/index de liste/attributs de containers, et échange leur valeur.

`is` *:*
Cet opérateur attend à gauche un objet quelconque et à droite un identifiant. Cet identifiant doit correspondre à un nom de type *(*`Integer`, `List`, *...)*. L'opérateur fonctionne également pour les containers. Pour n'importe quel container, on peut tester son type en évaluant `object is MyContainer`, et ce, même si la variable `MyContainer` existe et ne contient pas la constante `MyContainer`. L'opérateur `is` n'évalue pas l'identifiant donné à droite, il regarde simplement si le nom correspond au type de l'objet à gauche.

2.1.2 - Opérateurs unaires

`-` *:* Cet opérateur peut également être utilisé comme opérateur unaire. L'opérande se place à droite. Renvoie l'opposé d'un nombre.

`del` *:*
Cet opérateur prend une variable à droite, et supprime cette variable.

`@` *:*
Cet opérateur est un opérateur unaire, et attend une chaîne de caractères à droite. Il renvoie la valeur de la variable ayant pour nom la chaîne de caractères indiquée comme opérande.

`&` *:*
Cet opérateur est un opérateur unaire qui attend une variable à droite. L'opérateur renvoie le nom de la variable.

`++` *:*
Opérateur unaire, attend une variable, un index de liste ou un attribut de container à gauche. `a++` correspond exactement à `a += 1`.

`--` *:*
Opérateur unaire, attend une variable, un index de liste ou un attribut de container à gauche. `a--` correspond exactement à `a -= 1`.

`not` *:*
Opérateur unaire, opérande à droite. Effectue une négation logique.

2.1.3 - Opérateurs spéciaux

`parallel` *:*
Cet opérateur est un opérateur spécial qui ne prend pas en entrée un objet mais une expression. l'opérateur `parallel` doit recevoir à droite un appel de fonction utilisateur, et lance cet appel de fonction en parallèle, dans un nouveau fil d'exécution.

Exemple *:* `parallel f(arg1, arg2, arg3)` lance la fonction `f` en parallèle sur les arguments `arg1`, `arg2` et `arg3`. De plus amples explications seront données dans la section dédiée au multitâches.

`.` *:* Cet opérateur est également un opérateur spécial qui attend à gauche un objet et à droite un appel de fonction ou de méthode. Écrire `objet.fonction(arg1, arg2)` correspond exactement à `fonction(objet, arg1, arg2)`

`>>` : Cet opérateur est également un opérateur spécial qui attend à gauche un container et à droite un nom de champ de container.

2.1.4 - Les priorités opératoires

Les priorités de tous les opérateurs sont réparties parmi *9* niveaux de priorité de *0* à *8*. Le niveau le plus élevé correspond à l'opérateur le moins prioritaire, celui qui sera évalué en dernier.

Niveau *0* : `>>`
Niveau *1* : `-` (unaire)
Niveau *2* : `&`, `@`, `.`,
Niveau *3* : `**`, `++`, `--`, `EE`
Niveau *4* : `*`, `/`, `%`, `//`
Niveau *5* : `+`, `-`
Niveau *6* : `==`, `!=`, `<=`, `>=`, `<`, `>`, `in`
Niveau *7* : `and`, `or`, `xor`, `not`, `=>`, `parallel`
Niveau *8* : `=`, `+=`, `-=`, `*=`, `/=`, `<-`, `->`, `del`, `<->`

2.2 - Les fonctions

Les fonctions sont des objets qui produisent un comportement, souvent une sortie, et généralement à partir d'arguments. Si un objet `myFunction` est une fonction (que ce soit une fonction utilisateur ou une fonction built-in), la fonction peut être exécutée sur les arguments `a1`, ..., `an` via la syntaxe suivante : `myFunction(a1, a2, ..., an)`. Lorsque cette expression est évaluée, l'objet renvoyé est la sortie produite par la fonction. Certaines fonctions ne produisent pas de sortie (ou du moins ce n'est pas leur but), ces fonctions renvoient `None`.

Les fonctions peuvent évidemment être composées entre elles, et avec des opérateurs.

Partie *3* : Structure d'un programme

Comme dit dans l'introduction, Neon est un langage qui permet de décrire des programmes de calcul séquentiels. Cela signifie que le langage Neon permet de décrire l'exécution d'une succession de tâches simples. On appelle également cela un programme.

Un programme est un assemblage (concaténation et composition) de blocs de code. Un bloc de code représente une tâche à exécuter. Il y a beaucoup de types de blocs de code différents, chacun avec ses particularités.

Le bloc de code le plus basique est l'expression. Considérée en tant que bloc de code, une expression n'est pas importante pour le résultat qu'elle renvoie, mais les actions qu'elle produit pendant son évaluation. Quand on écrit une expression en tant que bloc de code, l'expression est évaluée, mais son résultat final est ignoré. Si on écrit le programme suivant, constitué d'un unique bloc de code qui est une expression :

```
2 + 3
```

la valeur *5* sera bel et bien créée, mais ignorée et supprimée. Même si cette expression est un bloc de code valide, elle ne sert à rien en tant que bloc de code car elle ne produit aucun comportement. L'état du programme (les variables, la mémoire) est le même après son évaluation et avant son évaluation.

En revanche, si on écrit le programme suivant, encore une fois constitué d'un unique bloc de code qui est une expression :

```
maVariable = 2 + 3
```

la valeur créée par l'expression *2 + 3* aura été utilisée puisque la variable `maVariable` la contient désormais. En revanche, la valeur renvoyée par l'expression entière `maVariable = 2 + 3` (`None`) est encore une fois ignorée.

Un programme Neon est une succession et une composition de blocs de code. Pour exécuter deux blocs de code à la suite, il faut soit les séparer d'un retour à la ligne, soit les séparer d'un point virgule `;`. Exemple d'un programme composé d'une succession de tâches simples :

```
maVariable = 2 + 3
maVariable *= 7
maVariable --
print(maVariable)
```

À partir de ces blocs de code basiques, il est possible d'en construire de plus complexes.

3.1 - Les blocs conditionnels

Les blocs conditionnels servent à exécuter du code à certaines conditions. Alors que les blocs de code constitués d'expressions à la suite s'exécutent quoi qu'il arrive, le code à l'intérieur de blocs conditionnels s'exécute uniquement si une certaine condition spécifiée est vraie.

Un bloc conditionnel complet s'écrit de la manière suivante :

```
if (expression booléenne) then
    code à exécuter
elif (autre expression booléenne) then
    autre code à exécuter
⋮ nombre arbitraire de blocs elif
elif (autre expression booléenne) then
    autre code à exécuter
else
    autre code à exécuter
end
```

Voici comment il est interprété : les expressions booléennes vont être testées une à une dans l'ordre. D'abord celle du `if`, puis celle du premier `elif`, etc. Pour la première de ces expressions qui s'évalue à `True`, le code juste en dessous est exécuté, et on sort du bloc conditionnel.

Si aucune des conditions ne s'évalue à `True` (donc toutes les conditions s'évaluent à `False`), le code à l'intérieur du `else` est exécuté.

Comme dit plus haut, ceci est un bloc conditionnel complet.

Un bloc conditionnel valide doit forcément commencer par un bloc `if`, puis peut contenir un nombre quelconque de bloc `elif` (y compris zéro), puis peut contenir un bloc `else`.

Les blocs conditionnels suivants sont valides :

```
if (1+1 == 2) then
    print("vrai")
end
```

```
if (1+1 == 0) then
    print("vrai")
else
    print("faux")
end
```

3.2 - Les boucles `while`

Les boucles sont des structures qui répètent l'exécution d'un certain bloc de code en fonction de différents paramètres.
Une boucle `while` répète l'exécution d'un bloc de code tant qu'une certaine condition est vraie. Voici la syntaxe d'une boucle `while` *:*

```
while (expression booléenne) do
    code à exécuter
end
```

Il est important de noter que la condition est toujours testée **avant** l'exécution du code dans le bloc.

On en déduit plusieurs faits *:*

- Le code à l'intérieur du bloc peut toujours supposer vraie la condition
- Si la condition est fausse dès l'entrée dans la boucle, le code ne sera jamais exécuté

3.3 - Les boucles `for`

Les boucles `for` permettent d'exécuter un bloc de code en faisant varier la valeur d'un entier, appelé variant, sur un certain intervalle.

La syntaxe complète d'une boucle `for` est *:*

```
for (variant, début, fin, pas) do
    code à exécuter
end
```

Les champs `début`, `fin` et `pas` doivent être des expressions d'évaluant en nombres entiers, et le champ `variant` doit être une variable.

À l'entrée dans la boucle `for`, la variable utilisée pour le variant va passer en variable locale à la boucle. C'est-à-dire que la valeur précédente de la variable ne sera plus accessible, et la variable aura la nouvelle valeur utilisée dans la boucle `for`. Quand la boucle `for` sera terminée, la valeur précédente du variant sera restaurée. Si la variable utilisée pour le variant n'était pas définie avant la boucle `for`, elle redeviendra indéfinie. La section *4.3* détaille plus les variables locales et globales.

Le comportement de la boucle `for` est simple *:* le code à exécuter à l'intérieur du bloc sera exécuté pour chaque valeur différente du variant, à commencer par `début`, et jusqu'à `fin` **exclu**, en incrémentant la variable de `pas` à chaque tour de boucle.

La variable utilisée pour le variant est réactualisée à chaque tour de boucle, ce qui signifie que même si elle est modifiée à l'intérieur du code, elle sera restaurée comme si de rien n'était à la fin du tour de boucle, et la boucle ne sera pas impactée.

Il n'est pas toujours obligatoire de spécifier le pas et la valeur de début de la boucle. Il existe deux autres manières de définir une boucle `for` *:*

```
for (variant, début, fin) do
    code à exécuter
end
```

Dans ce cas, le pas par défaut est de *1*.

```
for (variant, fin) do
    code à exécuter
end
```

Dans ce cas, le pas par défaut est de *1* et la valeur de début de la boucle est *0*.

3.4 - Les boucles `foreach`

Les boucles `foreach` sont basées sur le même principe que les boucles `for` : elles permettent d'exécuter un bloc de code pour chaque valeur d'une liste ou chaque caractère d'une chaîne de caractères, dans l'ordre croissant des indices.

Voici la syntaxe :

```
foreach (element, iterable) do
    code à exécuter
end
```

Comme pour les boucles `for`, la variable `element` est locale à la boucle et est restaurée à chaque nouveau tour de boucle.

3.5 - Instructions de contrôle

Les instructions de contrôle sont à elles seules des blocs de code, au même titre que les expressions. Elles doivent donc être séparées d'autres blocs de code par un retour à la ligne ou un point virgule.

3.5.1 - Instruction `break`

L'instruction `break` doit être utilisée exclusivement dans les boucles. Lorsque'une instruction `break` est rencontrée, l'exécution quitte immédiatement la boucle la plus intérieure dans laquelle se situe le `break`, et continue juste après ladite boucle.

3.5.2 - Instruction `continue`

L'instruction `continue` doit être utilisée exclusivement dans les boucles. Lorsque'une instruction `continue` est rencontrée, l'exécution saute au début de la boucle la plus intérieure dans laquelle se situe le `continue`. Dans cette opération, la condition de la boucle est revérifiée, donc si la condition était fausse au moment du `continue`, celui-ci aura le même effet qu'un `break`.

3.5.3 - Instruction `pass`

L'instruction `pass` ne sert strictement à rien.

3.6 - La gestion d'erreurs

Neon dispose d'un système de gestion d'erreurs. Lorsqu'une erreur survient, une exception est déclenchée. Il est possible de détecter le déclenchement d'exceptions grâce aux blocs `try ... except`, et d'exécuter du code en fonction de l'exception déclenchée.

Les exceptions existant par défaut sont détaillées dans la section *1.2.7*.

Il est possible de manipuler les exceptions en tant qu'objet, et d'en créer à l'aide de la fonction `createException`.

Il est également possible de déclencher volontairement des exceptions grâce à la fonction `raise` (voir comment l'utiliser dans la section *1.2.8)*.

Voici comment écrire un bloc `try ... except` :

```
try
    code à exécuter
except (Exception1, Exception2, ...) do
    code à exécuter
; nombre arbitraire de blocs except
except (Exception3, ...) do
```

```
    code à exécuter
end
```

Lorsqu'on exécute un bloc `try ... except`, le code à l'intérieur du `try` est d'abord exécuté. Si tout se passe bien (c'est-à-dire aucune exception n'est déclenchée), le bloc `try ... except` a terminé quand le code à l'intérieur du `try` a fini.

En revanche, si une exception est levée pendant l'exécution du code à l'intérieur du `try`, l'exécution du code sera stoppée au moment de l'exception, et le premier bloc `except` dont l'une des exceptions indiquées en tête correspond à l'exception lancée sera exécuté, entièrement et normalement. Quand ce bloc `except` a terminé, l'exécution du bloc `try ... except` est également terminée.

À noter que Exception, Exception2, ... doivent être des expressions s'évaluant en objets de type `Exception`. Ces expressions peuvent par exemple être des constantes correspondant aux exceptions (DivisionByZero, etc) ou encore des variables contenant des exceptions...

Le nombre d'exceptions que l'on peut spécifier pour un bloc `except` n'a aucune limite, et peut même être nul. Dans le cas où l'on ne spécifie aucune exception entre les parenthèses, le bloc `except` est exécuté quelle que soit l'exception levée.

Partie 4 : Définition de fonctions et méthodes

En plus des fonctions définies par défaut par l'interpréteur Neon (de type Built-in function), il est possibles de définir deux autres types de fonctions : les fonctions utilisateur (de type Function), et les méthodes (de type Method).

Une fonction est un bloc de code qui prend en entrée des arguments, qui produit une sortie ou un comportement, et renvoie éventuellement une sortie.

La définition de fonctions est ce qui doit permettre de rendre un code Neon le plus compréhensible possible, proche d'un texte en anglais.

4.1 - Définition de procédures

Les fonctions les plus simples conceptuellement sont les procédures. Ce sont des fonctions qui ne prennent aucun argument, et qui ne renvoient aucune sortie. Ces fonctions effectuent donc toujours la même action quand elles sont appelées. Définir de telles fonctions sert uniquement à rendre un code plus compréhensible et potentiellement plus concis, en faisant appel à des fonctions avec un nom clairement défini plutôt qu'exécuter directement un bloc de code.

Neon ne distingue pas les procédures des fonctions à part entière. Une procédure est simplement la manière de désigner les fonctions sans arguments qui ne renvoient rien. Une procédure se définit tout simplement comme suit :

```
function maProcedure() do
    code à exécuter
end
```

Une procédure renvoie `None`. Toutes les fonctions ne comportant pas de `return ()` renvoie `None`.

4.2 - Fonctions basiques

Voyons maintenant comment envoyer des arguments à une fonction, et renvoyer un résultat depuis une fonction.

```
function maFonction(arg1, arg2, arg3) do
    code à exécuter
end
```

Cette fonction prend trois valeurs en arguments, et ces valeurs sont stockées dans les variables `arg1`, `arg2` et `arg3`. Le code à l'intérieur de la fonction peut donc utiliser ces variables sachant que leur valeur est celle des arguments envoyés à la fonction.

Contrairement au code que l'on écrit à n'importe quel endroit dans un programme, le code d'une fonction peut utiliser un bloc de code supplémentaire *:* le bloc `return ()`. Ce bloc de code peut soit contenir une expression *:* `return (expression)`, soit rester vide *:* `return ()`. L'expression peut être de n'importe quel type. Lorsqu'un bloc `return ()` est rencontré dans une fonction, celui-ci met fin à la fonction. Cela signifie que tout code situé après un bloc `return ()` n'est jamais exécuté.

Lorsque le bloc `return ()` contient une expression, cette expression est évaluée au moment où on rencontre le `return ()`, et la valeur obtenue est renvoyée comme retour de la fonction.

Lorsque le bloc `return ()` est vide, la valeur renvoyée est simplement `None`.

Mettre un `return ()` ou un `return (None)` à la toute fin d'une fonction revient exactement à ne rien mettre.

4.3 - Variables locales et globales

Afin de ne pas interférer avec les variables d'un programme et de rendre invisible le code exécuté dans une fonction aux yeux du code appelant la fonction, il existe différents niveaux de localité de variables. Ces niveaux de localité impliquent que certaines variables peuvent avoir plusieurs valeurs différentes en même temps, dont une seule de ces valeurs n'est accessible.

En réalité, pendant un programme Neon, une variable est comme une pile. Lorsque l'on modifie une variable, lorsque l'on accède à sa valeur, on manipule toujours la valeur au sommet de la pile.

Lorsque'une variable devient locale à un nouveau bloc de code, une nouvelle valeur est ajoutée sur la pile, et c'est cette valeur que va manipuler le bloc de code. À la fin du bloc de code ayant utilisé la variable comme variable locale, la valeur utilisée est dépilée, et la précédente valeur redevient la valeur principale.

Les blocs de code ayant la capacité de rendre des variables locales sont *:*
- Les blocs conditionnels
- Les boucles `for`/`foreach`
- Les fonctions utilisateur
- Les méthodes

Lorsque l'on appelle une fonction, les variables utilisées comme arguments de cette fonction sont automatiquement transformées en variables locales à cette fonction. La valeur précédente de ces variables est donc préservée.

En plus des variables utilisées comme arguments, il est possible de rendre locale n'importe quelle variable locale grâce au bloc de code `local ()`. Ce bloc de code attend comme argument des variables, séparées par des virgues. Le nombre de variables n'est pas limité. `local (var1, var2, var3...)` rend local toutes les variables entre les parenthèses. Dans ce cas-là, leur valeur de ces variables sera restaurée à la fin du bloc de code le plus intérieur dans lequel était situé `local ()`.

Comme dit précédemment dans les sections *3.3* et *3.4*, les variables utilisées comme variant dans les boucles `for` et `foreach` sont aussi automatiquement rendues locales.

4.4 - Méthodes

Les méthodes sont des fonctions avec une fonctionnalité supplémentaire. Dans une fonction, si l'on modifie la valeur des variables utilisées pour récupérer les arguments, cela n'aura évidemment aucun impact sur les arguments eux-mêmes.

Dans le cas d'une méthode, c'est un peu différent.

À la fin d'une méthode, la valeur de la variable utilisée pour stocker le premier argument est automatiquement affectée à l'objet envoyé en premier argument. Ainsi, toute modification effectuée sur le premier argument à l'intérieur d'une méthode sera également effective à l'extérieur de la méthode.

À part cette particularité, les méthodes sont exactement comme les fonctions.

Pour définir une méthode, il suffit d'utiliser le mot-clé `method` au lieu du mot-clé `function` lors de la définition.

4.5 - Méthodes avancées de passage d'arguments

La manière classique d'envoyer des arguments à des fonctions est de séparer les expressions des arguments par des virgules *:* `fonction(exp1, exp2, exp3)`. En réalité il existe des fonctionnalités bien plus avancées.

4.5.1 - Passage d'arguments nommés

Parfois, certaines fonctions prennent en entrée beaucoup d'arguments, et il est difficile de se souvenir de l'ordre exact dans lequel spécifier les arguments. De plus, l'appel à de telles fonctions peut être assez complexe à relire *:* prenons l'exemple hypothétique de la fonction suivante *:* `function drawFilledRect(x, y, width, height, fg_r, fg_g, fg_b, fg_a, bg_r, bg_g, bg_b, bg_a)`.

C'est pour cela que l'interpréteur Neon fournit la possibilité de spécifier des arguments de fonction dans un ordre ne respectant pas celui de la définition de la fonction, en nommant directement les arguments que l'on spécifie.

Pour spécifier des arguments de cette manière, il suffit d'indiquer le nom de l'argument que l'on spécifie, de le suivre par `:=` puis par sa valeur.

Il y a une restriction néanmoins *:* les arguments nommés doivent obligatoirement être situés après tous les arguments positionnels.

Exemple avec la fonction suivante *:* `function f(a, b, c)`

Tous les appels suivants à la fonction `f` correspondent à l'appel classique `f(1,2,3)` *:*

```
f(a := 1, c := 3, b := 2)
f(1, 2, c := 3)
f(1, b := 2, 3) # Appel incorrect car l'argument `3` est donné après l'argument nommé
`2`.
```

4.5.2 - Arguments optionnels

Les arguments optionnels permettent de définir des fonctions dont il n'est pas nécessaire de spécifier tous les arguments lorsqu'on les appelle. Lorsque l'on définit un argument optionnel, on donne au moment de la définition de la fonction une valeur par défaut, à donner à cet argument dans le cas où l'appel à la fonction n'a pas donné de valeur à l'argument.

Lorsque l'on définit une fonction attendant des arguments classiques, on se contente d'écrire *:* `function maFonction(arg1, arg2, arg3)` en séparant par des virgules les noms des arguments.

Pour définir un argument optionnel, il suffit de suivre le nom de l'argument par `:=`, puis par l'expression qui s'évaluera en la valeur par défaut.

Exemple *:* `function maFonction(obligatoire1, obligatoire2, optionnel1 := expression...)`

Il y a ici également une restriction *:* les arguments optionnels doivent être tous définis après les arguments obligatoires.

4.5.3 - Nombre illimité d'arguments

Il est également possible de définir des fonctions attendant un nombre illimité d'arguments. Pour cela il faut utiliser l'opérateur spécial `...` à la place d'un nom d'argument.

Exemple : `function maFonction(arguments normaux, ...)`

Cette fois-ci, les `...` doivent réellement être écrits tels quels et ne sont par un raccourci d'écriture de ce document.

Lorsqu'une fonction peut recevoir un nombre illimité d'arguments, seules les valeurs n'ayant pas pu être affectées à des arguments normaux sont comptées dans les arguments supplémentaires.

Lors de l'appel à une fonction au nombre d'arguments illimité, une variable locale spéciale est créée. Cette variable est une liste contenant toutes les valeurs n'ayant pas pu être affectées à des arguments normaux (donc les valeurs comptabilisées dans le `...`), et est accessible sous le nom `__local_args__`. Si la variable `__local_args__` existait au moment de l'appel de la fonction, la valeur est écrasée.

4.5.4 - Arguments vraiment optionnels

Lorsque l'on définit une fonction attendant un nombre illimité d'arguments, il est également possible de définir des arguments après les `...`. Ces arguments doivent obligatoirement être optionnels, et on les appelle les arguments vraiment optionnels.

Exemple : `function f(a, b, ..., c := 5)`

Comme les `...` englobent toutes les valeurs envoyés à la fonction après les arguments normaux, la seule manière de donner une valeur à un argument vraiment optionnel est de la spécifier via la syntaxe `:=`.

4.6 - Programmation d'ordre supérieur, clôtures

Il est possible de tirer à profit la manière dont Neon évalue et définit les fonctions pour obtenir un comportement similaire aux clôtures que l'on retrouve dans la plupart des langages de programmation.

Pendant la définition d'une fonction, les clôtures permettent de sauvegarder la valeur des variables non locales à cette fonction, mais locales à des fonctions à l'intérieur desquelles est définie notre fonction.

Exemple :

```
function plus(valeur) do
    function maFonction(a) do
        return (a + valeur)
    end
    return (maFonction)
end
```

Cette fonction prend en argument un nombre et renvoie une fonction qui ajoute ce premier nombre à un autre nombre. Du moins, c'est ce que l'on aimerait que fasse cette fonction. Or, écrite telle quelle, cette fonction ne fonctionne pas. En effet, la fonction utilise la variable locale `valeur`, qui existe lors de la définition de la fonction `maFonction`, mais est détruite au moment ou la fonction `plus` renvoie sa valeur.

Les clôtures ont été inventées afin de corriger ce problème. Pour plusieurs raisons, Neon ne dispose pas d'un tel système. La première raison est la transparence. Neon ne sauvegarde/restaure pas la valeur de variables dans le dos des programmeurs. Ensuite, Neon dispose déjà d'un système relativement lourd de traitement des arguments, et il n'était pas nécessaire de l'alourdir davantage avec un système de clôtures. Cependant, la manière dont est codée Neon permettrait si le besoin s'en fait sentir, de facilement implémenter un tel système.

En revanche il est possible d'en simuler le comportement.

Ainsi, un appel à la fonction renvoyée par `plus` utilisera une variable non définie et déclenchera une erreur.

Cependant il est bel et bien possible de coder cette fonction en Neon, d'une manière un peu différente. Pour cela il est important de comprendre comment l'interpréteur définit les fonctions.

Il y a deux stades pour définir une fonction.

Le premier stade se situe au niveau de l'analyse purement syntaxique de la fonction, avant le lancement de l'exécution du programme. À ce stade-là, Neon enregistre le code de la fonction, détecte les arguments dont elle a besoin, regarde si ces arguments sont des arguments optionnels ou non, et finalement, crée un objet fonction partiel. Cet objet fonction est partiel car il ne contient pas encore la valeur par défaut des arguments optionnels. En effet, leur valeurs ne sont pas encore connues à ce stade-là.

Le deuxième stade se situe pendant l'exécution du programme, au moment où l'on passe le bloc de définition de la fonction. À ce stade-là, on peut évaluer la valeur par défaut des arguments optionnels, puis les enregistrer dans un objet fonction complet qui va être affecté à la variable dont le nom est celui de la fonction.

Ainsi, la valeur par défaut des arguments optionnels est évaluée une et une seule fois au moment de la définition d'une fonction, et enregistrée dans l'objet fonction. Il est donc possible d'avoir des fonctions différentes alors qu'elles proviennent du même bloc de définition de fonction, à cause du fait que les fonctions ont été définies à des moments différents et donc n'ont pas les mêmes valeurs par défaut pour leurs arguments.

En exploitant cette caractéristique, voici une version correcte de la fonction `plus` évoquée plus haut :

```
function plus (valeur) do
    function maFonction(a, valeur := valeur) do
        return (a + valeur)
    end
    return (maFonction)
end
```

Ici, nous utilisons les arguments optionnels pour capturer la valeur de la variable `valeur` au moment de la définition de la fonction. Lorsque la fonction renvoyée par `plus` sera appelée, comme l'argument optionnel ne sera pas utilisé, la variable `valeur` va automatiquement recevoir la valeur évaluée au moment de la définition de la fonction, celle qu'il fallait sauvegarder.

4.6 - Programmation modulaire

Lorsque l'on écrit des programmes de taille conséquente possédant différentes composantes bien définies, il est usuel de découper ce programme en modules. En Neon, un module est un ensemble de variables et de fonctions possédant le même préfixe.

4.6.1 - Le caractère `~`

Pour reconnaître toutes les fonctions et variables appartenant à un module, tous les noms de ces éléments doivent posséder le même préfixe de module. Un préfixe de module est de la forme : `NomCommencantParUneMajuscule~`.

Exemple : Pour créer trois éléments `a`, `b` et `c` appartenant au module `Module`, il suffit de les appeler `Module~a`, `Module~b` et `Module~c`.

Pour créer un module, il suffit de créer une variable ou une fonction dont le préfixe est le nom de ce module.

Cette syntaxe par préfixes facilite la reconnaissance des module par Neon. Ainsi, en tapant `help("modules")`, la fonction `help` listera tous les modules définis, et de la même manière en tapant `help("NomModule")`, la fonction `help` listera tous les objets dont le nom commence par `NomModule~`.

4.6.2 - La fonction `loadNamespace`

La fonction `loadNamespace` prend en argument une chaîne de caractères correspondant à un nom de module, et crée une copie de tous les objets de ce module en leur enlevant le préfixe de module. Ainsi, après un appel à `loadNamespace`, il n'est plus nécessaire d'écrire les préfixes de modules devant les noms des éléments du module chargé.

Cependant il est important de comprendre que `loadNamespace` n'agit que sur les éléments déjà définis au moment où elle est appelée.

4.6.3 - Surcharge de fonctionnalités

Neon fournit la possibilité de surcharger certains opérateurs, l'appel de fonctions ainsi que les fonctions d'affichage sur certains types de containers. Il est possible de configurer Neon pour que l'utilisation de certains opérateurs sur certains types de containers exécute une fonction Neon définie par l'utilisateur plutôt que l'opérateur original.

Les opérateurs surchargeables sont *:*

```
+ : add
- : sub
/ : div
* : mul
% : mod
// : eucl
** : pow
- (unaire) : minus
in : in
```

Pour surcharger un opérateur sur un certain type de containers `MonContainer`, il suffit de regarder le nom que porte l'opérateur dans la liste ci-dessus, et définir la fonction `MonContainer~nom`, avec `nom` le nom de l'opérateur à surcharger dans la liste ci-dessus. C'est cette fonction qui définira l'action effectuée par l'opérateur sur ces objets.

En plus des opérateurs, on peut surcharger la fonction `str` en définissant `MonContainer~str` et l'affichage *(*`print`, `output` et l'affichage dans la console*)* en définissant `MonContainer~repr`.

Il est également possible de surcharger l'appel de fonctions pour des containers. Pour cela il faut définir la fonction `MonContainer~call` qui doit prendre deux arguments *:* l'objet que l'on appelle, et la liste d'arguments de l'appel. Pour les containers implémentant la surcharge de l'appel de fonction, ils pourront être appelés comme n'importe quelle fonction, si ce n'est que tous les arguments sont des arguments positionnels *(*pas de possibilité de spécifier des arguments par leur nom*)*.

Si au moins l'un des arguments d'un opérateur est un container d'un type pour lequel cet opérateur est surchargé, la fonction de surcharge sera appelée.

4.6.4 - Mot-clé `import`

Le langage Neon fournit un bloc de code appelé `import ()` qui permet d'exécuter le contenu de programmes Neon à partir de leurs noms. Le mot-clé `import` ne fonctionne qu'avec des fichiers dont l'extension est `.ne`. Pour exécuter un fichier Neon dont l'extension est `.ne`, il suffit de mettre en argument du `import` une chaîne de caractères correspondant au nom du fichier sans l'extension.

Pour la version TI_EZ*80* de Neon, les fichiers Neon étant des AppVars sans extension, il faut écrire le nom entier du fichier pour l'importer.

`import` peut recevoir un nombre illimité d'arguments. En tant que bloc de code à part entière, il doit être séparé d'autres blocs de code par un retour à la ligne ou un point virgule, au même titre qu'un `return ()`.

Partie *5 :* Programmation concurrente

5.1 - Vision par processus

Dans les sections précédentes, quand on décrivait le comportement du programme, on en parlait comme d'une sorte de tête de lecture parcourant le programme et exécutant les instructions qu'elle rencontre. Cette vision correspond relativement bien au réel déroulement de l'interprétation d'un programme. Cette tête de lecture dont on parle implicitement quand on décrit le comportement d'un programme est en quelque sorte la personnification de l'exécution d'un programme.

À partir de maintenant et grâce à la programmation concurrente, nous allons créer des programmes possédant plusieurs têtes de lecture différentes, c'est-à-dire exécutant simultanément différentes portions d'un programme. On dira alors qu'un programme possède différents fils d'exécution.

Avant d'aller plus loin, il est essentiel de comprendre comment une telle chose est possible, et surtout comment elle est réalisée dans Neon.

De nombreux ordinateurs ou appareils électroniques ne disposent que d'un unique processeur avec un unique coeur. Cependant, la quasi totalité des systèmes d'exploitation nécessitent la capacité d'exécuter plusieurs programmes en même temps. Ces systèmes d'exploitation gèrent alors ce qu'on appelle de l'entrelacement entre processus. Le processeur exécute quelques instructions d'un processus, puis quelques instructions d'un autre processus, puis revient au premier processus, etc. Il alterne l'exécution entre tous les processus. Cette gymnastique donne l'illusion que ces processus sont exécutés en même temps.

Neon ayant pour but de pouvoir être exécuté sur n'importe quelle plateforme, il n'effectue aucune supposition sur le système d'exploitation. Ainsi, il ne peut pas utiliser de telles fonctionnalités d'entrelacement.

C'est pourquoi l'interpréteur Neon gère lui-même l'entrelacement entre ses propres processus. Cette caractéristique possède des avantages, mais également des inconvénients.

En effet, l'avantage n°*1* est la possibilité d'exécuter plusieurs programmes à la fois sur des appareils pour lesquels c'est impossible autrement. Le fait que l'interpréteur ait directement le contrôle sur l'entrelacement permet également de mieux contrôler certains paramètres comme le temps passé à exécuter un processus avant de passer au suivant.

En revanche, sur des appareils disposant de plusieurs coeurs d'exécution simultanée, l'interpréteur Neon sera incapable de distribuer les tâches sur ces différents coeurs.

Ainsi, un programme Neon écrit de manière concurrente ne sera jamais plus rapide que sa version écrite entièrement séquentiellement. Les fonctionnalités concurrentes de Neon ne doivent être utilisées que lorsque cela facilite l'écriture du programme.

5.2 - L'opérateur `parallel`

L'opérateur `parallel` permet de lancer une fonction en parallèle, c'est-à-dire de créer un nouveau fil d'exécution, un nouveau processus exécutant la fonction spécifiée avec les arguments spécifiés.

Cet opérateur est un opérateur unaire attendant à droite un appel à une fonction utilisateur.

Exemple *:* `parallel fonction(arg1, arg2, arg3)`

L'opérateur `parallel` appliqué à un appel de fonction utilisateur va créer un nouveau processus, ajouter ce processus dans la file d'attente *(*c'est-à-dire que son exécution commencera quand viendra son tour*)*, et renvoyer une promesse sur ce processus.

5.3 - Le retour de processus via les promesses

Les promesses sont les objets renvoyés par l'opérateur `parallel`, et sont de type `'Promise'`. Cet opérateur est l'unique manière d'obtenir des objets de type `'Promise'`.

Lorsqu'ils sont créés, tous les processus se voient affecter un identifiant. Cet identifiant est unique pour les processus en cours d'exécution, mais si un processus a terminé, son identifiant peut être réaffecté plus tard à un autre processus.

Le lancement d'un processus avec l'opérateur `parallel` va créer une unique promesse associée à ce processus grâce à son identifiant. Cette promesse peut être utilisée pour identifier ce processus par rapport aux autres.

Les promesses ne servent pas uniquement à identifier des processus, mais permettent également aux processus de renvoyer un résultat. En effet, un processus est créé à partir d'un appel de fonction. Quand le processus aura terminé,

il pourra donc renvoyer la valeur de retour de la fonction sur laquelle il a été appelé. Ce renvoi se fera via la promesse récupérée lors du lancement du processus.

Tant que le processus est en cours d'exécution, la promesse renvoyée est de type `"Promise"`. Une fois que le processus a terminé, toutes les promesses associées à ce processus (la promesse originale et celles obtenues à partir de la promesse originale) stockées dans des variables, des listes ou des containers vont instantanément se transformer en la valeur de retour du processus.

5.4 - Attente passive

Lorsque l'on programme à base de processus, il est fréquenr de créer des programmes qui attendent. Or, pour attendre, la seule manière de le faire est une boucle ressemblant à ceci :

```
while (not condition) do
    pass
end
```

Cette manière d'attendre est appelée de l'attente active, car le programme tourne activement pendant qu'il attend. Il est souvent relativement peu efficace d'attendre de cette manière ; de gaspiller du temps de calcul à ne rien faire. En effet ce genre de structure est pratiquement toujours utilisé avec de la programmation concurrente, et la condition ne peut devenir vraie que pendant l'exécution d'un autre processus.

C'est pourquoi Neon fournit une deuxième manière d'attendre, mais de façon passive. Cela signifie que si la condition permettant de mettre fin à l'attente n'est pas vérifiée, l'exécution passe directement au processus suivant. L'interpréteur favorise l'exécution des autres processus sur celui qui est en train d'attendre.

L'attente se fait grâce à la syntaxe `await(condition)`. L'exécution de `await(condition)` lance une attente passive **jusqu'à ce que la condition soit vraie**. La condition doit être une expression booléenne.

`await()` est un bloc de code au même titre que `import()` et `return ()`. Il doit être séparé d'autres blocs de code par un retour à la ligne ou un point virgule.

5.5 - Variables locales aux processus

Afin de ne pas interférer entre eux, les processus disposent chacun d'un contexte : chaque variable n'a potentiellement pas la même valeur dans tous les processus. Lorsqu'une variable est globale, elle a la même valeur dans tous les processus, et une modification de la variable par un processus sera visible dans tous les processus.

En revanche, à partir du moment où le code d'un processus localise une variable (en l'utilisant comme argument de fonction ou en faisant appel à `local`), le processus concerné manipulera une version personnelle de la variable. Quand la variable ne sera plus locale dans le processus, celui-ci manipulera de nouveau la version globale de la variable.

Comme les processus de Neon sont en réalité exécutés à la suite et non en parallèle, lorsqu'une variable est locale à un processus, ce dernier doit faire un travail de sauvegarde/restauration entre le moment où vient son tour et le moment où il doit passer la main à un autre processus.

Chaque processus possède une liste des variables qui lui sont locales. Au moment de reprendre son exécution, il restaure sa version locale de la variable, travaille dessus, puis remet la version globale de la variable en sauvegardant sa version locale au moment de passer au processus suivant.

5.6 - Blocs atomiques

L'entrelacement entre processus géré par l'interpréteur peut rendre imprédictible l'exécution de certaines séquences de code. En effet dans un programme, il est usuel d'effectuer des suppositions d'une ligne à l'autre en fonction des conditions déjà vérifiées. Or lorsque d'autres processus peuvent être exécutés à n'importe quel moment entre deux lignes, ce genre de code peut facilement devenir incorrect.

Les blocs atomiques permettent de s'assurer qu'une certaine séquence de code s'exécute de manière atomique, c'est-à-dire ne sera interrompue à aucun moment, et s'exécutera en une seule fois, sans qu'aucun autre code ne puisse être exécuté en même temps que le code dans le bloc atomique.

Un bloc atomique s'écrit de la manière suivante :

```
atomic
    code à exécuter
end
```

5.7 - Fonctions du système d'entrelacement

Le passage d'un processus à l'autre par l'interpréteur Neon n'est pas effectué à n'importe quel moment. Il est effectué par une fonction nommée `neon_interp_yield`. À chaque fois qu'un bloc de code est exécuté, un compteur est décrémenté, et lorsque ce compteur atteint zéro, la fonction `neon_interp_yield` est appelée, met en pause le processus actuel et reprent l'exécution d'un autre processus.

5.7.1 - La fonction `setAtomicTime`

Le nombre de fois que `neon_interp_yield` décrémente le compteur avant de passer au processus suivant est stocké dans la variable `ATOMIC_TIME`. C'est la valeur par défaut du compteur atomique d'un processus. Par défaut, `ATOMIC_TIME` vaut *1500*, c'est-à-dire que pour chaque processus, au bout de *1500* appels à la fonction `neon_interp_yield`, celle-ci passera au processus suivant.

La fonction `setAtomicTime` permet de modifier cette valeur *(1* est la plus petite valeur possible*)*.

Partie 6 : Fonctionnalités supplémentaires

6.1 - Arguments de programme

Lorsque l'on appelle un programme en ligne de commande, il est possible de lui envoyer des arguments séparés par des espaces. Lorsque Neon est utilisé en mode exécution *(*c'est-à-dire en envoyant un nom de fichier en premier argument de l'interpréteur*)*, les arguments suivants sur la ligne de commande sont récupérés par Neon puis stockés dans la liste `__args__`. Cette variable est ensuite accessible par le programme pour traiter les arguments.

6.2 - Variables et constantes prédéfinies

6.2.1 - La constante `Pi`

Neon possède une constante `Pi` avec pour valeur le nombre `3.141592653589793`. Cette valeur est arrondie en fonction de la précision offerte par les nombres flottants.

Certaines variables d'environnement sont prédéfinies par l'interpréteur afin de permettre au programme exécuté d'obtenir des informations sur son environnement.

6.2.2 - La variable `__name__`

Cette variable permet au programme de connaître le nom du fichier dans lequel il est écrit. Plus exactement, dans le programme principal *(*code situé dans le fichier lancé en ligne de commande*)*, la variable `__name__` vaut `"__main__"`. Dans un fichier importé, cette variable change de nom et vaut le nom du fichier. Lorsque l'exécution revient au fichier principal, la valeur de cette varieble redevient `"__main__"`.

6.2.3 - La variable `__platform__`

Cette variable permet de connaître le système d'exploitation et l'architecture pour lesquels l'interpréteur Neon utilisé est compilé. Les différentes valeurs possibles de cette variable sont `"LINUX_AMD64"`, `"WINDOWS_AMD64"` et `"TI_EZ80"`.

La valeur de cette variable est visible dans le texte affiché au lancement du mode console.

6.2.4 - La variable `__version__`

Cette variable est une chaîne de caractères représentant la version de l'interpréteur Neon utilisé. La valeur de cette variable est visible dans le texte affiché au lancement du mode console.

6.3 - La variable spéciale `Ans`

En mode console, à chaque fois qu'une expression est entrée dans le terminal, la variable `Ans` prend la valeur du résultat de cette expression.

Partie *7 :* Extensions non standard

En plus du corps du langage contenant tout ce qui est documenté jusqu'ici dans ce document et disponible sur toutes les plateformes, il est possible sur certaines plateformes d'importer des fonctions supplémentaires non disponibles sur les autres plateformes.

7.1 - L'extension graphique pour la plateforme TI_EZ*80*

Cette extension est mal nommée puisqu'elle n'est pas seulement graphique, mais permet à la fois de dessiner à l'écran, et de gérer les appuis du clavier.

Lors du chargement de l'interpréteur, rien de ce qui est défini dans cette extension n'est accessible, il faut au préalable exécuter `init(Graphics)` pour initialiser l'extension en mémoire.

Sur les plateformes qui ne supportant pas l'extension graphique, exécuter `init(Graphics)` lèvera l'exception `NotImplemented`.

Cette extension définit un ensemble de fonctions et de types de containers qui permettent de créer des objets graphiques et de les afficher.

La manière dont fonctionne la préfédinition de types de containers est assez spéciale, il est important de comprendre comment cela fonctionne.

En effet, comme expliqué plus tôt dans cette documentation, il est normalement suffisant *d'écrire* un container d'un certain type pour que ce type soit défini. Il n'est jamais nécessaire de *définir* explicitement un type de container.

Le type est défini en fonction du premier objet de ce type rencontré par l'interpréteur.

Le problème est que certaines fonctions graphiques prennent en argument des objets graphiques (cercles, rectangles, lignes) qui suivent une définition bin précise, avec des champs précis, etc. Pour que les fonctions graphiques puissent reconnaître de manière efficace si un objet est un cercle, un triangle, etc, tous ces objets graphiques sont prédéfinis lors du chargement de l'extension.

Les types de containers définis lors de l'initialisation du module `Graphics` sont :

- `Point(x, y)` : `x` et `y` de type `Integer` ou `Real`
- `Circle(x, y, r, c, f)` : `x`, `y` et `r` de type `Integer` ou `Real`, `c` de type `Integer` et `f` de type `Bool`
- `Rect(x, y, w, h, c, f)` : `x`, `y`, `w` et `h` de type `Integer` ou `Real`, `c` de type `Integer` et `f` de type `Bool`
- `Line(x0, y0, x1, y1, c)` : `x0`, `y0`, `x1` et `y1` de type `Integer` ou `Real` et `c` de type `Integer`
- `Text(t, x, y, fg, bg, s)` : `t` de type `String`, `x` et `y` de type `Integer` ou `Real`, `fg`, `bg` et `s` de type `Integer`
- `Triangle(x0, y0, x1, y1, x2, y2, c)` : `x0`, `y0`, `x1`, `y1`, `x2` et `y2` de type `Integer` ou `Real`, et `c` de type `Integer`
- `Polygon(p, c)` : `p` est une liste de containers de type `Point`, et `c` est un entier
- `Ellipse(x, y, a, b, c, f)` : `x`, `y`, `a`, `b` et `c` de type `Integer` et `f` de type `Bool`
- `FloodFill(x, y, c)` : `x` et `y` de type `Integer` ou `Real` et `c` de type `Integer`

Cela signifie qu'après avoir initialisé le module graphique, tous les containers dont les noms sont listés au-dessus devront posséder les paramètres données au-dessus. Les types des attributs de sont pas forcés de coincider pour que la définition du container soit correcte, mais les types données ici sont les types attendus dans les champs des containers pris en argument par les fonctions graphiques.

Le module graphique peut être initialisé alors que certains des types décrits ci-dessus ont déjà été définis, mais si les types déjà définis ont une définition différente de celle attendue ici, une erreur sera levée.

Voici une description un peu plus explicite de l'utilité des différents champs de ces objets.

- Les champs `x`, `y`, `x0`, `y0`, ... représentent des coordonnées de pixels
- Les champs `a` et `b` des ellipses correspondent respectivement au rayon horizontal et au rayon vertical de l'ellipse
- Les champs `c`, `fg` et `bg` sont des couleurs. `fg` est la couleur des lettres lorsque l'on dessine du texte, et `bg` est la couleur de remplissage autour de chaque lettre.
- Le champ `f` *(*filled*)* indique si la figure doit être tracée en remplissant son contour ou non. Il faut noter que les triangles sont automatiquement remplis, et que les polygones ne sont pas remplis.
- Les champs `w` et `h` *(*width, height*)* représentent respectivement la longueur et la largeur des objets. Par exemple pour dessiner un rectangle, `width` et `height` correspondent à la largeur et la hauteur du triangle. Les champs `x` et `y` du rectangle sont les coordonnées du coin supérieur droit du rectangle à dessiner.
- `r` *(*radius*)* est le rayon du cercle
- `s` *(*size*)* est la taille des caractères dessinés. Un paramètre `s` à *1* correspond à un texte de taille basique. Pour `s` = *2*, la hauteur des lettres est doublée. Pour `s` = *3*, la hauteur et la largeur des lettres sont doublées. Pour `s` = *4*, la hauteur des lettres est triplée et la largeur est doublée. La logique est similaire pour la suite. Les valeurs de `s` impaires correspondent aux lettres dont les dimensions ont été uniformément multipliées par un ratio, et les valeurs paires correspondent à la taille impaire précédente, à laquelle on a uniquement étiré les lettres en hauteur. Il n'est pas nécessaire de comprendre tout cela, il suffit de comprendre que plus `s` est grand, plus le texte l'est aussi.

Voyons maintenant à quoi servent tous ces objets.

7.1.1 - L'écran

L'écran de la TI-*83* Premium CE / Edition Python *(*ou de la TI-*84* Plus CE*)* est un rectangle de *320* pixels de large et *240* pixels de haut.

Le système de coordonnées place l'origine *(*x=*0*, y=*0)* tout en haut à gauche. Ainsi, le pixel au coin inférieur gauche est de coordonnées *(*x=*0*, y=*239)*, le pixel au coin supérieur droit est de coordonnées *(*x=*319*, y=*0)* et le pixel au coin inférieur droit est de coordonnées *(*x=*319*, y=*239)*.

Comme vous l'avez remarqué lors des définitions d'objets dans la section précédente, les couleurs sont des entiers, entre *0* et *255*.

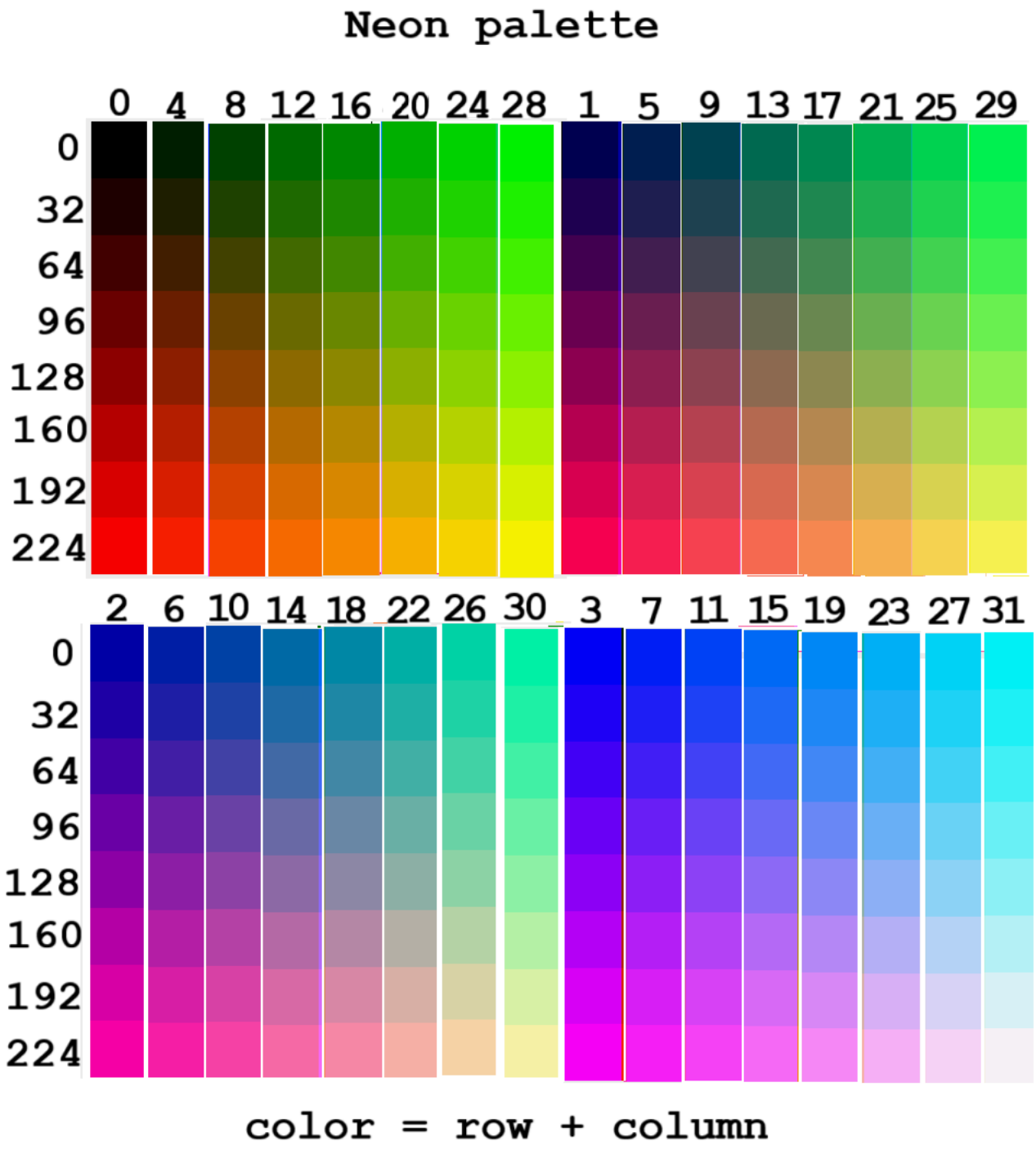
Chaque couleur est codée au format rgb sur *1* octet *(8* bits*)* de la manière suivante *:*
- *3* bits pour le rouge
- *3* bits pour le vert
- *2* bits pour le bleu

Ainsi, il y a *8* teintes de rouge et de vert différentes et *4* teintes de bleu différentes. Toutes les combinaisons donnent exactement *256* couleurs différentes.

Parce qu'il n'est pas pratique de calculer `(r/36) << 5 | (g/36) << 2 | (b/85)` à chaque fois que l'on a besoin d'une couleur, il existe une fonction `rgb` pour convertir des couleurs du format RGB *(*intensités entre *0* et *255)* au format de Neon sur un octet.

Pour le dessin de texte *(*l'objet `Text`*)*, la couleur *255* représente la couleur transparent.

Voici néanmoins une représentation de toutes les couleurs accessibles en Neon *:*



La fonction `rgb`

Les couleurs disponibles pour les graphiques sont des entiers entre *0* et *255*. Cette représentation n'est pas très commode, aussi, la fonction `rgb` permet d'associer à une couleur RGB sa couleur en Neon. Cette fonction fait évidemment des approximations puisqu'il est impossible de stocker entre *0* et *255* autant de couleurs que *256*256*256*.

La fonction `rgb` prend en argument trois entiers entre *0* et *255* correspondant respectivement aux niveaux de rouge, vert et bleu.

La fonction `draw`

La fonction `draw` est la fonction principale permettant de dessiner des choses à l'écran. Cette fonction permet d'afficher d'un coup un nombre illimité d'objets quelconque.

Le nombre et le type des arguments attendus par cette fonction sont parfaitement libres, tout peut être envoyé à cette fonction. Si un objet envoyé à `draw` est directement affichable (container de type Rect, Line, etc) alors il sera affiché sur l'écran. Si un objet envoyé à `draw` est une liste ou un container non affichable, tous les objets affichables qu'il contient (peu importe à quelle profondeur dans l'objet) seront affichés. Les objets de type `Integer`, `Bool`, etc sont ignorés. Si un objet envoyé à `draw` contient une fonction (built-in ou définie par l'utilisateur), la fonction sera exécutée.

De manière générale, la seule chose à retenir pour savoir utiliser cette fonction est : `draw` va toujours essayer de dessiner ce qui est dessinable, peu importe jusqu'où il faut chercher les objets dessinables dans les arguments qu'on lui donne.

Souvent lorsque l'on dessine plusieurs objets à l'écran, l'ordre de dessin est important. Certains objets doivent être par-dessus d'autres objets, d'autres tout au fond, etc. La fonction `draw` dessine toujours les objets de la gauche vers la droite.

Exemple :
Supposons qu'on veuille coder un jeu de course de bateaux. Dans ce cas, on utilisera une liste `bateaux` dans laquelle on stocke tous les bateaux.

Utilisons par exemple ce type de container pour représenter les bateaux : `Bateau(vitesse, nom, forme)`. Le champ `vitesse` est un nombre indiquant la vitesse du bateau et `nom` contient le nom du joueur.

Le champ `forme` contient une liste d'objets affichables permettant de dessiner le bateau (exemple : un rectangle rempli pour le contour, un triangle pour l'avant, et quelques lignes pour montrer la traînée derrière le bateau).

Dans un jeu utilisant de tels objet, il suffit pour dessiner tous les bateaux d'appeler `draw(bateaux)`.

Le comportement de la plupart des objets lorsqu'on les dessine est assez clair : `Circle` trace un cercle, `Rect` dessine un rectangle, `Line` dessine une ligne, etc.

L'objet `FloodFill` quant à lui effectue un remplissage d'une certaine zone de l'écran dont les coordonnées sont spécifiées.

La fonction `setPixel`

La fonction `setPixel` permet de changer directement la couleur d'un pixel.

`setPixel(x, y, color)` met à `color` la couleur du pixel d'abscisse `x` et d'ordonnée `y`.

- `x` doit être de type `Integer` ou `Real`
- `y` doit être de type `Integer` ou `Real`
- `color` doit être de type `Integer`

La fonction `getPixel`

La fonction `getPixel` permet de récupérer la couleur d'un pixel. `getPixel(x, y)` renvoie la couleur du pixel d'abscisse `x` et d'ordonnée `y`. `x` et `y` doivent être de type `Integer` ou `Real`.

La fonction `setTextTransparentColor`

Lorsque l'on dessine un objet `Text`, on doit spécifier `fgcolor` et `bgcolor` les couleurs de premier plan et d'arrière plan du texte.

La fonction `setTextTransparentColor` permet de spécifier une couleur qui sera utilisée *en tant que transparent*. Par exemple, si on appelle `setTextTransparentColor` avec une certaine couleur, et qu'on utilise ensuite cette couleur en tant que paramètre `bgcolor`, le texte ne sera pas surligné.

Par défaut, la couleur utilisée pour le transparent est la couleur `255`.

La fonction `getTextWidth`

Cette fonction calcule la largeur en pixels d'un objet `Text` donné, en prenant en compte le paramètre `size`.

La fonction `menu`

Cette fonction permet d'afficher un menu interactif pour choisir entre différentes options.

Elle prend trois paramètres :
- Le titre du menu (une chaîne de caractères)
- La liste des items à afficher (une liste de chaînes de caractères)
- Une chaîne de caractères à afficher dans le cas où la liste d'items est vide

La fonction `menu` renvoie l'élément choisi (donc une chaîne de caractères), ou `None` si l'utilisateur n'a rien choisi.

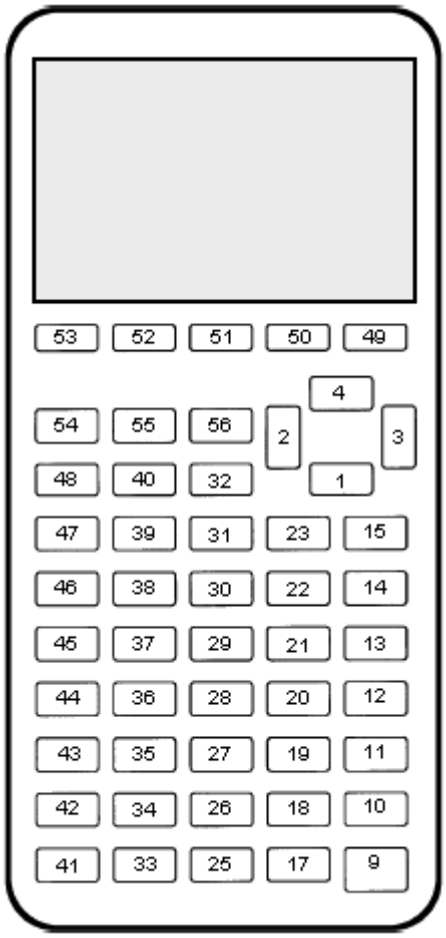
7.1.1 - Le clavier

L'accès au clavier est géré par une seule fonction : la fonction `getKey`, qui ne prend pas d'arguments.

Cette fonction renvoie un entier correspondant à la dernière touche pressée, ou zéro si aucune touche n'a été pressée depuis le dernier appel à `getKey`.

Chaque touche de la calculatrice possède un code unique permettant de l'identifier. Le code associé à chaque touche est montré sur le schéma ci-dessous.

La touche ON *(41)* n'est pas utilisable avec `getKey`. L'appui sur cette touche déclenche l'erreur **`KeyboardInterrupt`**.



Conclusion

Cette documentation se veut être une description exhaustive des fonctionnalités du langage de programmation Neon. Si vous pensez qu'il manque des informations, que des informations sont fausses, pour une quelconque remarque/question ou encore pour signaler un bug, n'hésitez pas à rejoindre le serveur Discord de Neon : `https://discord.gg/wkBdK35w2a`.

Vous pouvez également me contacter par mail à l'adresse contact@langage-neon.raphaael.fr.