Le langage CBOT

1. Résumé des fonctions de CBOT

Le langage CBOT fait partie intégrante de CoLoBoT. Il permet de programmer les robots.

1.1. Instruction

Une instruction décrit une opération à faire, chaque instruction se termine par un point-virgule.

1.2. Bloc d'instructions

Plusieurs instructions groupées dans un bloc délimité par { et }.

2. Déclaration des fonctions

Une fonction est définie selon cette syntaxe :

```
extern public void object :: Nom ( liste de paramètres )
{
      instructions;
}
```

extern et **public** sont optionnels, **void** peut être remplacé par le type du résultat retourné par la fonction (par un **return**), **object ::** est optionnel également.

2.1. extern

La présence de ce mot signale à CoLoBoT que le nom de la fonction déclarée ici doit être mis dans le bouton correspondant à ce programme. Il ne faut donc mettre qu'un seul **extern** pour chaque script.
De plus, la fonction déclarée **extern** doit retourner **void** et n'avoir aucun paramètre :

```
extern void object :: Nom ( )
{
      instructions;
}
```

2.2. public

La présence de ce mot rend la fonction disponible partout dans le niveau. Cette fonction peut alors être utilisée dans le programme d'un autre robot, par exemple. Une autre fonction ne peut pas alors avoir le même nom ailleurs.
Si le robot qui contenait la fonction déclarée public est détruit, les autres robots qui utilisent cette fonction vont stopper sur une erreur !

2.3. void, int, float, boolean, string

Type de résultat que la fonction retourne. **void** signifie qu'il n'y a pas de résultat retourné. Les autres types correspondent aux déclarations de variables (voir plus loin).
Il est possible de retourner un tableau de valeur, par exemple en mettant **int [ ].**

2.4. object ::

La présence de cette chaîne devant le nom d'une fonction permet de dire que la fonction utilise les éléments de la classe **object**. Ce mot permet donc d'accéder aux caractéristiques (**position**, **temperature**, **energyCell**, ...) du robot appelant cette fonction. En l'absence de ce mot, la procédure n'a pas connaissance de ces informations.

```
public boolean TEST(int valeur)
{
      return valeur > temperature; // erreur à la compilation
}
```

Par exemple, cette procédure TEST n'a pas accès à "temperature" qui appartient à "object".

2.5. Liste de paramètres

Une fonction peut recevoir des données en entrée. Il faut en donner la liste, avec à chaque fois le **type** de la variable et le **nom** qui lui est donné.

```
void exemple ( int a, float y, boolean test, string s )
```

La procédure exemple, ici, va recevoir un nombre entier (a), un nombre réel (y), un booléen (test) et une chaîne (s).
Note: il n'est pas possible d'attribuer des valeurs par défaut à ces paramètres.
Il est possible de déclarer une fonction avec le même nom, si les paramètres sont différents en type :

```
void exemple ( string s )
```

Est donc possible en plus de la première déclaration. Lors de l'appel à la fonction, CBOT recherche la fonction dont les paramètres correspondent au mieux.

3. Variables

Les variables permettent d'enregistrer des informations durant l'exécution du programme. Les variables doivent être déclarées selon leur type avec un nom et une éventuelle valeur initiale. Des variables d'un même type peuvent être déclarées en plaçant leurs noms séparés par des virgules. Les tableaux de variable peuvent être déclarés en plaçant des crochets [ ] soit après le type, soit après le nom.

3.1. int

Déclare un nombre entier.

```
int   nb;               // variable nb pour l'instant non initialisée
int   a = 3, b;         // a initialisé à 3, b non initialisée
int   [ ] table;        // un tableau de nombre entier
int   t1[3], t2[4][2];  // t1, tableau de 3 nombres entiers
                        // t2, tableau de 4 * 2 nombres entiers
```

3.2. float

Déclare un nombre réel (décimal).

```
float val;                  // variable val, non initialisée
float x = 5.3, y = -2.2;    // x et y initialisés
float [ ] table;            // un tableau de nombres réels
flaot xx[5][2], yy[3];      // tableau de 5*2 et tableau de 3
```

3.3. boolean ou bool

Déclare un nombre booléen. Un tel nombre ne peut valoir que **true** ou **false** (vrai ou faux).
bool est équivalent à **boolean**.

```
boolean     res;            // variable res pour l'instant non initialisée
bool        i = true, b;    // i initialisé à vrai, b non initialisée
bool        [ ] table;      // un tableau de booléens
bool        t1[3], t2[4][2]; // deux tableaux de booléens
```

3.4. string

Déclare une chaîne de caractères.

```
string      s;              // variable s pour l'instant non initialisée
string      t, txt = "text"; // t non initialisée, txt valant "text"
string      [ ] table;      // un tableau de phrases
string      t1[3], t2[4][2]; // deux tableaux
```

3.5. point

Le type **point** n'est pas un type de base du langage mais une extension définie par CoLoBoT.
Un point est composé de trois valeurs réelles (x, y, z). Il s'agit d'une **classe intrinsèque**, c'est à dire qu'on ne travaille pas avec des pointeurs, mais avec des instances directement.

```
point A;                // un point indéfini
point B(3,5), C(1,2,3); // deux points initialisés
A = B ;                 // fait une copie de B dans A
A.x = 0;                // ne modifie pas B.x
```
Note: de nouvelles classes intrinsèques ne peuvent pas être définies par l'utilisateur.

3.6. object

Le type **object** n'est pas non plus un type de base du langage, mais une extension définie par CoLoBoT. Il s'agit d'une **classe** gérée par des pointeurs. La description des éléments de cette classe est la suivante :

Type	Nom	Contenu
int	category	Catégorie de l'objet (par exemple TitaniumOre).
point	position	Position (x;y;z) de l'objet.
float	orientation	Orientation en degrés.
float	pitch	Inclinaison avant/arrière de l'objet.
float	roll	Inclinaison latérale de l'objet.
float	energyLevel	Niveau d'énergie (0..1).
float	shieldLevel	Niveau du bouclier (0..1).
float	temperature	Température du réacteur (0..1).
float	altitude	Altitude par rapport au sol.
float	lifeTime	Age de l'objet en secondes (à partir de la version 1.7).
object	energyCell	Objet pile.
object	load	Objet transporté.

Selon la catégorie de l'objet, certains éléments n'ont pas de sens. Par exemple :

Catégorie	Elément	Description
Titanium	shieldLevel	Un cube de titanium n'a pas de bouclier.
WheeledGrabber	temperature	Un robot à roues n'a pas de réacteur.
WheeledGrabber	energyLevel	Un robot n'a pas d'énergie. En revanche, sa pile a un niveau d'énergie (donc energyCell.energyLevel).

Une instance de cette classe est créé pour chaque objet (robots, bâtiments, minerais, …). L'instance du robot courant est disponible si la fonction est déclarée avec **object ::** devant le nom.
Même s'il est possible de créer une instance de la classe **object** avec la fonction **new**, cela n'a pas vraiment de sens (**new** ne va pas ajouter un robot).

```
{
      object chose = new object;    // nouvelle instance
      chose.position = this.position; // mémorise la position actuelle
      chose.load = this.load; // et le pointeur à l'objet transporté
      move(100);
      goto (chose.position);  // revient à la position de départ
}
```

3.7. null

Valeur particulière pour un pointeur, signalant que le pointeur n'est pas lié à une instance.

3.8. nan

Valeur particulière pour un nombre, signalant que le nombre n'est pas valable.

3.9. true

L'un des états (vrai) que peut prend une variable de type boolean.

3.10. false

L'un des états (faux) que peut prend une variable de type boolean.

4. Les opérateurs

4.1. if (condition) { instructions si vrai } else {instructions si faux }

condition est n'importe quelle instruction rendant un résultat de type **boolean**.
La partie **else** est optionnelle.
Si le résultat est vrai, le premier bloc d'instruction sera exécuté.
Si le résultat est faux, c'est le bloc éventuel après **else** qui sera exécuté.

4.2. condition ? expression1 : expression2

Evalue l'une ou l'autre des expressions, selon la condition.

```
int a = ( 4 > 3 ) ? 12 : 33;        // a prend la valeur 12
int b = ( 3 > 4 ) ? 12 : 33;        // b prend la valeur 33
```

4.3. while (condition) { instructions }

Répète le bloc d'instructions tant que la condition est vraie. Si la condition est fausse dès le départ, les instructions ne sont pas interprétées.

4.4. do { instructions } while (condition)

Exécute le bloc d'instructions tant que la condition est vraie. Contrairement au **while**, le bloc est exécuté au moins une fois.

4.5. for (avant ; condition ; fin) { instructions }

Cette structure de boucle permet d'exécuter plusieurs fois les instructions comprises dans le bloc. L'instruction **avant** est exécutée avant le premier tour de boucle. La **condition** détermine s'il faut continuer la boucle. Elle est examinée avant chaque tour de boucle, y compris le dernier. L'instruction **fin** est exécutée à la fin de chaque tour de boucle.
L'instruction **for** est rigoureusement équivalente à l'exemple suivant, utilisant **while** :

```
avant;
while ( condition )
{
      instructions;
      fin;
}
```

4.6. break

Permet d'interrompre l'exécution d'une boucle (**while**, **do**, **for**, **switch**) pour poursuivre à la suite du bloc.

```
while ( true )
{
      instruction1;
      break ;
      instruction2;
}
instruction3;
```

Dans cet exemple, instruction1 va être exécutée, puis instruction3. Instruction2 est sauté par la présence du **break**.

4.7. break label

Lorsque plusieurs boucles sont imbriquées, il est possible de spécifier laquelle doit être abandonnée. Il faut pour cela donner un nom (un label) aux boucles.

```
first :
for ( int i = 0 ; i < 10 ; i++ )
{
      second :
      while ( true )
      {
            if ( i++ == 4 ) break second;       // stoppe le while
            if ( i++ == 8 ) break first;        // stoppe le for
      }
}
```

4.8. continue

Permet de reprendre l'exécution d'une boucle (**while**, **do**, **for**) au début du bloc (si la condition reste vraie).

```
int i = 0;
while (i++ < 4)
{
      instruction1;
      if ( i < 2 ) continue;
      instruction2;
}
```

Dans cet exemple, le programme va exécuter instruction1, instruction1, instruction2, instruction1, instruction2, instruction1, instruction2. La première instruction2 est sautée par le continue.
Dans une boucle **for**, l'incrémentation est effectuée avant le test et avant de reprendre la boucle.

```
for ( int i = 0 ; i < 4 ; i++ )
{
      instruction1;
      if ( i < 2 ) continue;
      instruction2;
}
```

Dans cet exemple, le programme va exécuter instruction1, instruction1, instruction1, instruction2, instruction1, instruction2. Les deux premières instruction2 sont sautées par le continue. La différence avec l'exemple précédent, c'est que i est incrémenté après avoir exécuté le bloc et non avant.

4.9. continue label

Lorsque plusieurs boucles sont imbriquées, il est possible de spécifier laquelle doit être reprise. Il faut pour cela donner un nom (un label) aux boucles.

```
first :
for ( int i = 0 ; i < 10 ; i++ )
{
      second :
      while ( true )
      {
            if ( i++ == 4 ) continue second;    // reprend le while
            if ( i++ == 8 ) continue first;     // reprend le for
      }
}
```

4.10. return value

Une fonction peut retourner une valeur à la fonction appelante. La valeur doit être du type déclaré devant le nom de la fonction.

```
boolean TEST( float a, float b )
{
      return a > b;     // retourne un booléen
}
```

Ce qui permet d'utiliser ce résultat, par exemple avec

```
{
      if ( TEST( 3,4 ) ) message( "jamais !" );
}
```

4.11. switch / case / default

Le bloc **switch** permet de faire un branchement à certains endroits (case) selon une valeur entière.

```
int valeur = 2;
switch ( valeur )
{
      case 1 :
      instruction1;
      case 2 :
      instruction2;
      case 3 :
      instruction3;
      default :
      instruction4;
}
```

Ce programme va exécuter instruction2 puis instruction3 puis instruction4. instruction1 est sautée puisque la valeur 2 branche sur case 2. Une fois le branchement effectué, les autres instructions sont exécutées comme elles viennent. Si on ne veut pas que ce soit le cas, il faut utiliser la fonction **break.**

```
int valeur = 2;
switch ( valeur )
{
      case 1 :
      instruction1;
      break;
      case 2 :
      instruction2;
      break;
      case 3 :
      instruction3;
      break;
      default :
      instruction4;
}
```

la condition **default** est sélectionnée si aucun **case** au dessus ne correspond à la valeur.

4.12. try { instructions } catch (exception) {instructions} finaly {instruction}

Le bloc **try**, permet d'exécuter un bloc d'instructions pouvant provoquer une exception, de manière à traiter cette exception lorsqu'elle arrive.

```
int n;
try {
      n = 3/0;    // exception division par zéro
      instruction1;
}
catch (CBotErrZeroDiv)
{
      instruction2;
}
finaly
{
      instruction3;
}
```

Dans cet exemple, la division par zéro ne va pas interrompre le programme. Par contre, instruction2 va être exécutée. instruction1 quand à elle sera ignorée. instruction3 sera exécutée de toute manière qu'il se produise ou non une exception, traitée ou non par un bloc **catch**.

Liste des exceptions susceptibles de survenir lors de l'exécution d'un programme :

Exception	Signification
CBotErrZeroDiv	Division par zéro.
CBotErrNotInit	Variable non initialisée.
CBotErrBadThrow	Throw d'une valeur négative.
CBotErrNoRetVal	Fonction n'a pas retourné de résultat.
CBotErrNoRun	Run( ) sans fonction active.
CBotErrUndefFunc	Appel d'une fonction qui n'existe plus.
CBotErrNotClass	Cette classe n'existe pas.
CBotErrNull	Pointeur null.
CBotErrNan	Calcul avec un NAN.
CBotErrOutArray	Index hors du tableau.
CBotErrStackOver	Dépassement de la pile.
CBotErrDeletedPtr	Pointeur à un objet détruit.

4.13. try { instructions } catch (condition) {instructions} finaly {instruction}

La différence par rapport à l'instruction **try** du paragraphe précédent, c'est que l'on peut attraper une condition quelconque alors qu'il n'y a pas eu d'exception en fait.

```
try
{
      goto (cible.position );
      message ( " J'y suis " );
}
catch ( temperature > 0.6 )
{
      message ( " J'ai chaud ! " );
      return;
}
```

Malheureusement, ceci ne peut pas vraiment être exploité, car si le programme détecte bel et bien que le réacteur chauffe, le programme s'arrête, mais le robot poursuit l'opération **goto** en cours. Il n'y a pas moyen d'interrompre cette opération.

4.14. throw exception ;

Permet de provoquer une exception (erreur d'exécution) qui consiste en une valeur entière positive quelconque. Le programme est alors interrompu, le contrôle étant donné à un éventuel **catch** correspondant à cette exception.

5. Les classes

Le langage CBOT permet des déclarations de classes, limitées dans certains points.

5.1. class

La syntaxe est la suivante :

```
class NomDeLaClasse
{
      déclarations;
}
```

La classe est d'office publique (disponible de partout). Les éléments de la classe sont également publiques (disponible à tous). Les mots clés **extends** et **super** sont réservés, mais il n'est pas possible (actuellement) de créer une classe dérivées d'une autre classe. Le mot clef **static** est réservé, mais il n'y a pas de variables statiques.

Les déclarations peuvent être des déclarations de variables ou des déclarations de primitives (avec le bloc d'exécution), par exemple :

```
class MaClasse
{
      int   a, b;
      float x = 3.33;
      boolean MaPrimitive( int param )
      {
            return (param < a);
      }
      string      s = "hello";
}
```
Comme le montre cet exemple, il est possible d'initialiser la valeur des éléments par défaut (x = 3.33), ce qui rend le constructeur inutile. Toutefois, il est possible de définir un constructeur, en créant une procédure (de type **void**) ayant le même nom que la classe.

Il est également possible de définir plusieurs primitives ayant le même nom, mais avec des paramètres différents (ce qui est aussi valable pour les fonctions de base).
```
class MaClasse
{
      int   a,b;
      void  MaClasse( )
      {
            a = 2;  b = 3;
      }
      void MaClasse( int a, int b )
      {
            this.a = a;  this.b = b;
      }
}
```
Cet exemple déclare deux constructeurs pour MaClasse, l'un sans paramètre, l'autre avec deux paramètres. Comme les paramètres ont été nommés avec le même nom que les éléments a et b, il est nécessaire d'utiliser **this.a** et **this.b** pour accéder aux éléments de l'instance *(une solution plus simple consiste à donner des noms différents pour les paramètres)*.

Les constructeurs sont appelés automatiquement à la définition d'un élément de la classe.
```
void Test ( )
{
      MaClasse    elem1();          // constructeur sans paramètre
      MaClasse    elem2( 12, 13 );  // constructeur avec 2 paramètres
      MaClasse    elem3;            // pas de constructeur, elem3 == null;
}
```

5.2. new

Une instance de classe peut également être déclarée par l'opérateur **new.**
```
void Test ( )
{
      MaClasse element = new MaClasse;          // const. sans paramètre
      MaClasse autre = new MaClasse(12, 13);    // const. avec 2 paramètres
}
```

5.3. Pointeurs

CBOT travaille avec des pointeurs aux instances de classes et aux tableaux. Une instance est un élément physique contenant les informations liées à la classe. On peut comparer cela à une petite valise avec des conteneurs correspondants à la description de la classe. A chaque fois que l'on a besoin d'une nouvelle valise pour y mettre quelque chose, il faut créer une nouvelle instance. Les pointeurs, quant à eux, peuvent être considérés comme les porteurs de ces valises, à cela prêt qu'il y a souvent plusieurs porteurs pour la même valise.
Le fait qu'une instance à une classe soit créé automatiquement (**new** sous-entendu) ne change pas ce concept. On peut donc copier un pointeur dans un autre et avoir plusieurs pointeurs vers une même instance.

```
{
      MaClasse elem1();       // crée un pointeur (elem1) vers une
                              // nouvelle instance de MaClasse
      MaClasse elem2;         // crée un pointeur nul (sans instance)

      elem1 = null;           // détruit l'instance à MaClasse

      elem2 = new MaClasse(); // crée une nouvelle instance
      elem1 = elem2;          // pointe la même instance
      elem2 = null;           // ne détruit pas l'instance
                              // (utilisée par elem1)
      elem1 = new MaClasse(); // supprime l'ancienne instance pour mettre
                              // une nouvelle.
}
```

En passage de paramètre, c'est toujours le pointeur (une copie du pointeur) qui est donné à la procédure :

```
void TEST( MaClasse elem )
{
      elem.a = 12;            // modifie dans l'instance d'origine
      elem = new MaClasse();  // crée une nouvelle instance et met le
                              // pointeur (local) dessus
      elem.a = 33;            // modifie dans l'instance locale
}
```

Un appel à cette fonction va donner cela :

```
{
      MaClasse    toto;
      TEST( toto );
      message( toto.a );      // toto.a vaut 12
}
```

En effet, l'instance avec le résultat 33 n'était pointée que par elem dans la procédure TEST. A la sortie de TEST, le pointeur elem disparaît et l'instance qu'il pointait n'a plus de raison d'être.

Une procédure peut rendre une instance en sortie, pour cela il faut la définir ainsi :

```
MaClasse    TEST2 ( )
{
      MaClasse elem = new MaClasse();
      elem.x = 33;
      return MaClasse;
}
```

L'appel à cette fonction se fait alors ainsi :

```
{
      MaClasse toto = null;
      toto = TEST2();
      message( toto.a );      // toto.a vaut 33
}
```

5.4. Nettoyage

CBOT n'a aucun besoin de la fonction **delete** ni d'un "ramasse-miettes". Les instances créées avec **new** sont tout simplement détruites dès qu'il n'y a plus aucun pointeur qui y réfère ! Ceci est également valable avec les éléments des tableaux.

```
void exemple()
{
      MaClasse    pointeur = null;
      {
            MaClasse    elem1 = new MaClasse();
            MaClasse    elem2 = new MaClasse();
            pointeur = elem2; // copie le pointeur
      }
      // elem1 n'était connu que dans le bloc précédent
      // elem1 est donc supprimé, ainsi que l'instance qui lui est associée
      // de même elem2 est supprimé, mais comme pointeur pointe encore
      // l'instance, celle-ci est conservée
}
// pointeur est détruit, ainsi que l'instance associée.
```

6. Les tableaux

CBOT gère des tableaux de N dimensions, mais limités volontairement à un maximum de 9999 éléments par dimension. Les tableaux peuvent être de chaque type de base et même avec des éléments d'une classe quelconque. Pour déclarer un tableau, il faut mettre des crochets **[ ]** après le type, ou/et après le nom de la variable déclarée.

6.1. Tableau de nombres

```
int         [ ]   a;          // un tableau de nombre entier
int         a [ 12 ];         // idem limité à 12 éléments
float       [ ] z [ 5 ];      // 5 pointeurs à des tableaux de nombre réels
float       z [ 5 ] [ ];      // strictement équivalent à ci-dessus.
string      s1[3], s2[4];     // un tableau de 3 chaînes
                              // et un autre de 4 chaînes.
```

En vérité, lors de la déclaration d'un tableau, CBOT attribue juste un **pointeur nul** initialement.

```
int   a[5];       // a est un pointeur nul
```

Dès que l'on met une valeur dans le tableau, CBOT crée les éléments nécessaires et initialise le pointeur.

```
a[2] = 213;       // a est un pointeur sur 3 éléments [0], [1] et [2]
```

Après cette opération, **a** contient un pointeur vers les éléments du tableau. Les éléments [0] et [1] sont créés (non initialisés) car un tableau ne peut pas avoir d'éléments vides.

Le fait que l'on travaille avec des pointeurs ne doit pas être oublié. Voici quelques exemples :

```
int   a[3], b[7];
a[2] = 321;
b[5] = 123;
a = b;      // copie le pointeur au tableau b dans a, l'ancien contenu
            // du tableau a ([2]=321) est détruit par cette opération.
```

Suite à cette opération, **a** et **b** sont deux pointeurs vers le même tableau (celui qui contient [5] = 123). Le fait que **a** était déclaré initialement avec un maximum de 3 éléments n'a plus cours, **a** est désormais un pointeur vers un tableau pouvant avoir 7 éléments.

```
void TEST( int [ ] table )
{
      table [ 4 ] = 234;
}
```

Si l'appel à cette fonction est fait ainsi :

```
{
      int   a[ ];       // pointeur nul
      TEST ( a );
      message ( a[4] ); // pointeur toujours nul !
}
```

La fonction TEST reçoit un pointeur (table) initialisé à nul (valeur de **a**).
table[4] = 234; crée bel et bien un nouveau tableau dont le pointeur est mis dans table, mais la valeur nul de **a** n'est pas changée (table n'est pas **a**).
Le résultat correct peut être obtenu en faisant l'appel ainsi :

```
{
      int   a[ ];       // pointeur nul
      a[0] = 0;         // pointeur non nul
      TEST( a );
      message( a[4] );  // affiche 234
}
```
Ici, comme table reçoit une copie du pointeur **a** non nul, l'opération table[4] = 234 va inscrire cette valeur dans le tableau existant, sans créer un nouveau tableau.

6.2. Taille limitée

Lorsqu'un tableau est déclaré avec une taille maximale, le programme donne une erreur lors de l'exécution si on essaie de mettre un élément de trop. Aucune erreur n'est signalée lors de la compilation même si l'erreur paraît évidente.
```
{
      int   a [ 5 ];
      a [ 7 ] = 123;    // pas d'erreur à la compilation
                        // mais erreur lors de l'exécution
}
```

6.3. Tableaux d'objets

Déclarer un tableau d'objets permet de mémoriser dans ce tableau les **pointeurs** à des éléments de la classe **object**.
```
{
      object      tableau [ ];           // tableau = null
      tableau [ 0 ] = this;              // mémorise le pointeur à moi-même
      tableau [ 1 ] = radar(SpaceShip);  // et le pointeur au vaisseau
      tableau [ 2 ] = radar(WayPoint);   // pas trouvé, place null
}
```
Ce genre de tableau contient donc que des **pointeurs** vers des objets. Des éléments de la classe intrinsèque **point** réagissent différemment.
```
{
      point table [ ];             // table = null
      table[0] = position;         // copie la position
      table[1] = point( -34, 125); // copie un point
}
```
Ce tableau contient des copies des éléments. Par exemple, table[0] ne va pas changer même si le robot se déplace (donc **position** change).

6.4. sizeof (tableau)

La fonction **sizeof** permet de connaître le nombre d'éléments contenu dans le tableau.
En fait, c'est le dernier élément existant plus 1 (les cases "vides" sont comptées).
```
{
      int   a[12];
      a[5] = 345;
      message ( sizeof ( a ) );     // affiche 6
}
```
Dans cet exemple, il y a 6 éléments dans le tableau après avoir fait a[5] = 345. Les éléments non initialisés [0], [1], [2], [3] et [4] sont comptés avec l'élément [5].

7. Opérations

Les opérations permettent d'obtenir un résultat à partir d'un ou de deux opérandes.

7.1. variable = valeur

Assigne un résultat dans une variable. Le résultat doit être de type compatible.

```
a = 12;
str = "Hello";
pos = this.position;
```

7.2. *valeur1 + valeur2*

Additionne deux valeurs entre elles. Si l'une des deux valeurs est une chaîne de caractère, l'autre est convertie également en chaîne.

```
a = 12 + 5;
str = 12 + "Hello";                 // résultat est "12Hello"
str = 12 + 5 + "Hello" + 12 + 5;    // résultat est "17Hello125"
```

Dans ce dernier exemple, 12 et 5 sont additionnés puisque ce sont deux nombres, puis le résultat est converti en chaîne pour être ajouté à "Hello", ensuite la chaîne "17Hello" est ajoutée à l'élément suivant (12) donnant "17Hello12" et finalement le dernier élément est ajouté à la suite de cette chaîne pour avoir "17Hello125".
Une chaîne peut également être ajoutée à un élément d'une classe :

```
str = "Je suis à " + this.position;
```

7.3. *valeur1 - valeur2*

Soustrait une valeur d'une autre :

```
a = 12 - 5;
```

Peut également servir à changer le signe d'une valeur :

```
a = -b;
```

7.4. *valeur1 * valeur2*

Fait le produit entre deux valeurs :

```
a = 12 * 5;
```

7.5. *valeur1 / valeur2*

Divise une valeur par une autre. Si le diviseur est nul, le programme s'arrête sur une erreur.

```
a = 12 / 5;
```

7.6. *valeur1 % valeur2*

Rend le reste d'une division de valeur1 par valeur2.

```
int reste = 12 % 5;     // le reste est 2
```

7.7. *valeur1 ** valeur2*

Met une valeur à une puissance.

```
a = 3 ** 4;       // a vaut 81
```

7.8. *variable += valeur*

Ajoute une valeur au contenu actuel de la variable. On peut également ajouter des chaînes de cette manière.

```
int a = 5;
a += 3;     // a vaut maintenant 8
```

7.9. *variable -= valeur*

Soustrait une valeur au contenu actuel de la variable.

```
int a = 5;
a -= 3;     // a vaut maintenant 2
```

7.10. *variable *= valeur*

Multiplie le contenu actuel de la variable par une valeur.

```
int a = 5;
a *= 3;     // a vaut maintenant 15
```

7.11. *variable /= valeur*

Divise le contenu actuel de la variable par une valeur.

```
float a = 5;
a /= 3;     // a vaut maintenant 1.6666
```

7.12. *variable %= valeur*

Donne le reste de la division du contenu actuel de la variable par une valeur.

```
int a = 5;
a %= 3;     // a vaut maintenant 2 (reste de la division)
```

7.13. ++

Incrémente une valeur entière ou réelle. L'incrément peut se faire après ou avant l'utilisation de la variable.

```
i = 2;
message ( i++ );        // affiche 2 (puis i prend la valeur 3)
message ( ++i );        // affiche 4 (l'incrément s'est fait avant)
```

7.14. --

Décrémente une valeur entière ou réelle. Le décrément peut se faire après ou avant l'utilisation de la variable.

```
i = 7;
message ( i-- );        // affiche 7 (puis i prend la valeur 6)
message ( --i );        // affiche 5 (le décrément s'est fait avant)
```

7.15. >

Compare deux nombres et rend true si le premier est plus grand que le second.

```
bool result = a > b;
```

7.16. >=

Compare deux nombres et rend true si le premier est plus grand ou égal au second.

```
bool result = a >= b;
```

7.17. <

Compare deux nombres et rend true si le premier est plus petit que le second.

```
bool result = a < b;
```

7.18. <=

Compare deux nombres et rend true si le premier est plus petit ou égal au second.

```
bool result = a <= b;
```

7.19. ==

Compare deux nombres et rend true si le premier est égal au second.

```
bool result = a == b;
```

7.20. !=

Compare deux nombres et rend true si le premier est différent du second.

```
bool result = a != b;
```

*7.21. condition1 **and** condition2*
*condition1 **&&** condition2*

Fait un "et" logique entre deux valeurs de type boolean.
condition2 n'est pas calculée si condition1 est false (le résultat étant false quelle que soit condition2).

*7.22. condition1 **or** condition2*
*condition1 **||** condition2*

Fait un "ou" logique entre deux valeurs de type boolean.
condition2 n'est pas calculée si condition1 est true (le résultat étant true quelle que soit condition2).

*7.23. **not** condition*
__!__ condition

Rend l'inverse de la condition (faut devient vrai et réciproquement)

8. Opérations pour spécialistes

CBOT offre également une série d'opérations agissant bit à bit, les voici :

8.1. ~ valeur

Fait la **négation** de chaque bit de valeur.

*8.2. valeur1 **&** valeur2*

Fait le **et logique** entre chaque bits de valeur1 et valeur2.

*8.3. valeur1 **|** valeur2*

Fait le **ou logique** entre chaque bits de valeur1 et valeur2.

*8.4. valeur1 **^** valeur2*

Fait le **ou exclusif** entre chaque bit de valeur1 et valeur2.

*8.5. valeur1 **>>** valeur2*

Décale les bits de valeur1 vers la droite de valeur2 positions, le bit de signe est mis à zéro.

*8.6. valeur1 **>>>** valeur2*

Décale les bits de valeur1 vers la droite de valeur2 positions, le bit de signe est recopié.

*8.7. valeur1 **<<** valeur2*

Décale les bits de valeur1 vers la gauche de valeur2 positions.

*8.8. variable **&=** valeur*

Fait le **et logique** entre chaque bits du contenu de variable et valeur.

*8.9. variable **|=** valeur*

Fait le **ou logique** entre chaque bits du contenu de variable et valeur.

8.10. variable >>= valeur

Décale les bits du contenu de variable vers la droite de valeur positions, le bit de signe est mis à zéro.

8.11. variable >>>= valeur

Décale les bits du contenu de variable vers la droite de valeur positions, le bit de signe est recopié.

8.12. variable <<= valeur

Décale les bits du contenu de variable vers la gauche de valeur positions.

9. Préséances

Les opérations peuvent être combinées dans une même instruction pour en tirer un résultat. L'ordre dans lequel les opérations s'effectuent est déterminé selon la priorité des opérandes. Par exemple :

```
int a; float x;
x = a = 12 + 5 / 2 - 2;
```

Les opérations sont faites ici dans cet ordre :

- 5 / 2 donne 2.5
- 12 + 2.5 donne 14.5
- 14.5 -2 donne 12.5
- a = 12.5 donne 12 dans a
- x = a donne 12.0 dans x

Les opérandes sont évalués selon cette priorité :

1	+ -	opérations unaires (-14)
2	**	puissance
3	* / %	multiplication, division et modulo
4	+ -	adition et soustraction
5	>> >>> <<	décalages
6	> >= < <=	comparaisons plus grand ou plus petit que
7	== !=	comparaisons égal ou différent
8	&	and
9	^	xor
10	|	or
11	&&	logical and
12	||	logical or
13	= += -= *= /= %=	

Les opérations sur un même niveau sont évaluées de gauche à droite, sauf les assignations qui sont faites de droite à gauche.

10. Autres fonctions spécifiques à CoLoBoT

10.1. Fonctions mathématiques

10.1.1. sin (valeur)

Sinus d'un angle exprimé en degrés.

```
float a=sin(30)  // a vaut 0.5
```

10.1.2. cos (valeur)

Cosinus d'un angle exprimé en degrés.

```
float a=cos(120)  // a vaut -0.5
```

10.1.3. tan (valeur)

Tangente d'un angle exprimé en degrés.

```
float a=tan(45)  // a vaut 1
```

10.1.4. asin (valeur)

Arc-sinus d'un angle exprimé en degrés.

```
float a=asin(0.5)  // a vaut 30
```

10.1.5. acos (valeur)

Arc-cosinus d'un angle exprimé en degrés.

```
float a=acos(0.5)  // a vaut 60
```

10.1.6. atan (valeur)

Arc-tangente d'un angle exprimé en degrés.

```
float a=atan(1)  // a vaut 45
```

10.1.7. sqrt (valeur)

Racine carrée.

```
float a=sqrt(36)  // a vaut 6
```

10.1.8. pow (x, y)

Elévation àune puissance. Le résultat est x puissance y.

```
float a=pow(2, 8)  // a vaut 256
```

10.1.9. rand ( )

Retourne une valeur aléatoire comprise entre 0 et 1.

```
float a=rand( )  // a vaut 0.40287, par exemple
```

10.1.10. abs (valeur)

Retourne la valeur absolue d'un nombre.

```
float a=abs(-4)  // a vaut 4
float b=abs(29)  // b vaut 29
```

10.2. Fonctions spéciales

Les fonctions spéciales telles que **move**, **turn**, **goto**, etc. sont décrites en détail dans l'aide "online" de CoLoBoT. Nous ne décrivons ici que quelques particularités.

10.2.1. cmdline (rang)

Permet de récupérer l'un des paramètres spécifiés dans le descriptif de la scène par **cmdline=** et les valeurs séparées par des points-virgules. Le rang doit être compris entre 0 et 9. Supposons que le fichier sceneNN.txt contienne :

```
CreateObject pos=283;230 cmdline=283;230 type=AlienAnt ...
```

Alors, le programme CBOT pourra faire :

```
point center;
center.x = cmdline(0);
center.y = cmdline(1);  // center vaut (283;230;0)
```

10.2.2. produce (pos, angle, type, script)

Crée un nouvel objet. Cette instruction est généralement utilisée par la reine mère (AlienQueen) pour pondre des insectes. Les insectes AlienAnt, AlienSpider, AlienWasp et AlienWorm sont donc automatiquement créés dans des œufs (AlienEgg). Il est aussi possible de l'utiliser pour créer des ressources (TitaniumOre, UraniumOre, Titanium, PowerCell, NuclearCell, etc.). Par exemple :

```
produce(position, 0, AlienAnt, "%user%\ant01.txt");
```

Paramètre	Type	Signification
pos	point	Position de l'objet à créer.
angle	float	Orientation de l'objet à créer.
type	int	Type de l'objet à créer. Par exemple AlienAnt
script	string	Nom du programme initial contenu dans le dossier \script. Si le nom est précédé de %user%\, le programme sera cherché dans le dossier user\.

10.2.3. radar (type, angle, focus, min, max, sens)

Le premier paramètre est normalement le type de l'objet cherché. Il est possible de remplacer ce type par un tableau. L'instruction **radar** cherche alors l'un des types contenus dans le tableau. Par exemple, pour cherche un robot déménageur quelconque :

```
int   list[], i=0;
list[i++] = WingedGrabber;
list[i++] = TrackedGrabber;
list[i++] = WheeledGrabber;
list[i++] = LeggedGrabber;
item = radar(list, 0, 360, 0, 200);
```

10.2.4. flatground (center, rmax)

Nouvelle fonction, disponible à partir de la version 1.7 de CoLoBoT. Retourne le rayon de la surface plate (donc constructible) compris entre 0 et rmax.

10.2.5. abstime ( )

Nouvelle fonction, disponible à partir de la version 1.7 de CoLoBoT. Retourne le temps absolu en secondes depuis le début de la mission.

10.2.6. ipf (valeur)

Choix du nombre d'instructions élémentaires effectuées à chaque frame (**ipf** = instructions per frame). Il faut donner un nombre compris entre 1 et 10'000, la valeur par défaut étant 100. Avec une petite valeur, le programme CBOT effectuera peu d'instructions chaque seconde, mais occupera très peu le processeur. Attention avec les grandes valeurs, car le processeur sera très occupé, ce qui risque de baisser le frame rate, ou même de provoquer des saccades.

10.2.7. ismovie ( )

Indique si la mission est dans une phase de film, par exemple pendant l'atterrissage du vaisseau spatial (SpaceShip). Ceci peut être utile afin qu'un insecte attende la fin de l'atterrissage avant d'attaquer :

```
while ( ismovie() != 0 )  wait(1);
```

11. Equipe de développement

- Daniel Roux
- Denis Dumoulin
- Otto Kölbl
- Michael Walz
- Didier Gertsch

11.1. Beta testeurs

- Adrien Roux
- Didier Raboud
- Nicolas Beuchat
- Joël Roux
- Michael Jubin
- Daniel Sauthier
- Nicolas Stubi
- Patrick Thévoz

11.2. Copyright

11.3. Développeur

EPSITEC SA
Mouette 5
CH-1092 Belmont

colobot@epsitec.ch
www.colobot.com