SCOL SERVER

Version 4

TUTORIEL DU LANGAGE SCOL

Table des matières

1. PRESENTATION

Le présent chapitre vous aidera à faire les tous premiers pas en SCOL. Il contient une présentation de l'environnement de programmation SCOL.

1.1. Présentation de la machine virtuelle SCOL.

La machine virtuelle SCOL est le programme à la base de la technologie SCOL. Sa version Windows s'appelle **UsmWin.exe** (mis pour 'Universal SCOL Machine for Windows'). On la trouve habituellement dans '*C:/Program Files/SCOL/*'.

La machine virtuelle, comme son nom l'indique, crée une version virtuelle d'une machine 'idéale' dont les caractéristiques seraient les suivantes :

- la gestion de la mémoire est automatique : le développeur n'a pas à réserver ni à libérer lui-même la mémoire.
- le 'langage machine' de cette machine virtuelle est le langage SCOL.
- la gestion des communications réseau est complètement intégrée, puisqu'elle est masquée par le langage SCOL
- un certain nombre de librairies sont présentes, elles permettent de connecter la machine à une interface graphique simple d'emploi, à des interfaces sonores, et bien d'autres qui seront détaillées plus loin.

Comme toute machine, la machine virtuelle SCOL se nourrit de programmes, écrits en 'langage machine', ici en langage SCOL, ce qui signifie que la machine virtuelle inclut également un compilateur du langage SCOL. La machine virtuelle est dite 'universelle' car elle est le cœur de tout système fondé sur la technologie SCOL : un tel système est composé d'un nombre variable de machines SCOL qui communiquent entre elles.

Développer en SCOL revient donc à écrire des programmes dans le langage SCOL et à les donner à exécuter par une ou plusieurs machines SCOL. Il est important de noter que plusieurs machines virtuelles SCOL peuvent fonctionner sur un même ordinateur. Ce sera d'ailleurs toujours le cas, car une machine SCOL spéciale, appelée 'SCOL Voy@ger' fonctionnera toujours en tâche de fond de votre système.

Pour ceux que les détails techniques intéressent, précisons que le langage SCOL est compilé 'à la volée' vers un byte-code qui, lui, est interprété.

1.2. Lien entre la machine virtuelle SCOL et les fichiers de votre ordinateur

Comme toute machine, la machine SCOL a besoin d'une mémoire de masse pour stocker les programmes et les données des applications écrites en SCOL. La machine peut donc lire et écrire des fichiers.

Ceci est habituellement considéré par les spécialistes comme un problème grave de sécurité : les applications SCOL étant le plus souvent des applications connectées sur internet, l'utilisateur ne souhaite pas que le contenu de ses fichiers soit lu et transmis vers un autre point du globe. C'est pourquoi le système de gestion de fichiers de SCOL est spécialement étudié pour isoler d'une part les applications qui n'utilisent pas SCOL de celles qui utilisent SCOL, et d'autre part les applications SCOL entre elles.

Ce système, qui sera détaillé plus loin, repose sur la notion de '**partition SCOL'**. Une partition SCOL est un répertoire (ainsi que tous les sous-répertoires associés) de votre disque utilisable par SCOL : une machine SCOL ne pourra pas accéder d'elle-même à un fichier qui n'est pas présent dans une partition SCOL. Une machine SCOL peut utiliser plusieurs partitions SCOL : elles sont utilisées dans l'ordre de leur définition. Un fichier qui ne serait pas trouvé dans la première partition serait cherché dans la deuxième et ainsi de suite. L'écriture se fait toujours dans la première.

Par défaut, les partitions de votre machine SCOL sous Windows sont définies de la manière suivante :

c:/program files/SCOL/cache

c:/program files/SCOL/partition

Lorsque vous développerez vos premiers programmes, seule la seconde sera utilisée. Le principe est le suivant : la seconde partition est votre répertoire de travail, tandis que la première contient tous les fichiers que vous avez glanés en surfant de site en site ; vos fichiers sont donc protégés de vos pérégrinations.

Les partitions sont définies dans le fichier **usm.ini** (dont le chemin est normalement *c:/program files/SCOL/usm.ini*). La syntaxe de ce fichier sera définie plus loin.

1.3. Mise en oeuvre de l'environnement

Pour devenir un 'vrai' développeur SCOL, vous devez d'abord installer votre environnement de développement sur votre ordinateur. Pour cela, passez les étapes suivantes.

1.3.1. Installation de SCOL

Avant toute chose, vous devez installer SCOL sur votre machine. Pour cela, téléchargez la dernière version de la machine virtuelle sur le site de Cryo-Networks : **www.cryo-networks.com**. Elle est gratuite et pèse environ 1,3 Mo.

1.3.2. Configuration spéciale

La machine SCOL peut produire des fichiers rendant compte de son fonctionnement. Ces fichiers sont appelés '*fichiers de log*'. La création de ces fichiers ralentit le fonctionnement de la machine, c'est pourquoi, par défaut, ils ne sont pas produits. Pourtant, ces fichiers renferment des informations particulièrement intéressantes pour les développeurs. Notamment, ils indiquent les erreurs de syntaxe et, plus généralement, les erreurs de compilation. De plus, certaines fonctions du langage SCOL permettent d'écrire dans les fichiers de log en cours d'exécution. Ceci permet souvent un débuggage efficace de vos programmes.

Il faut donc commencer par réactiver la création des fichiers de log. Pour cela :

- lancez SCOL
- cliquer sur l'icône SCOL dans la barre des taches.
- ouvrir le menu " **Avancé**"
- ouvrir le menu " **Mode Expert** "
- remplacer la ligne `'echo 0'` par `'#echo 0'` (le caractère # permet de commenter la ligne)
- remplacer la ligne `'log no'` par `'log yes'`
- cliquer sur le bouton **Ok**

Ceci peut également se faire en éditant le fichier **usm.ini** (normalement placé sous *Windows* dans *C:\program Files\SCOL*).

Voila votre système configuré et vous êtes prêt à développer votre premier programme.

Lorsque vous voudrez désactiver la création des fichiers de log, il vous suffira de redonner à ces deux lignes leur état d'origine.

1.3.3. Pour développer

Pour développer en SCOL, il ne vous manque qu'une chose, un éditeur qui vous permettra de taper vos programmes.

Un seul conseil, utilisez votre éditeur favori. De plus, si cet éditeur offre un système d'indentation automatique pour le langage C, vous pouvez activer cette fonctionnalité. La syntaxe du langage SCOL a été conçue pour profiter de cette fonctionnalité bien répandue.

Si vous n'avez pas d'éditeur favori, vous pouvez utiliser le '**notepad'** inclus dans *Windows* .

2. HELLO WORLD

Nous commencerons par un premier programme bien connu. Il s'agit d'afficher le message '*hello world*' à l'écran. Nous donnerons plusieurs versions de ce programme.

L'idée de ce chapitre est de vous présenter quelques exemples de programme écrits en SCOL. Si certains points vous paraissent obscurs ou insuffisamment détaillés, soyez patient. Tout sera expliqué dans le chapitre suivant.

2.1. Première version

Pour commencer, créez un répertoire '*Tutorial*' dans votre répertoire '*SCOL/partition*'.

Puis tapez le fichier '*Tutorial/hello1.SCOL*' :

```
_load "Tutorial/hello1.pkg"
main
```

Tapez maintenant le fichier '*Tutorial/hello1.pkg*' :

```
/* commentary */
fun main()=
 _showconsole;
 _fooS ">>>>>>>>>>>> Hello World";;
```

A présent, double-cliquez sur le fichier *Tutorial/hello1.SCOL* pour exécuter le programme.

Une fenêtre blanche, finement rayée apparaît, avec un certain nombre d'inscriptions encore incompréhensibles. Les dernières lignes sont :

```
exec : _load "Tutorial/hello1.pkg"

loading C:\Program Files\SCOL\partition\Tutorial/hello1.pkg ...
typechecking
fun main : fun [] S
bytecode producing
loading complete
exec : main
>>>>>>>>>>>> Hello World
WRITE 9
exec : _connected
_connected : commande introuvable
WRITE_OK
>>end TimeOut
```

Vous voyez apparaître la mention '`>>>>>>>>>>>> Hello World`'. Cette fenêtre est appelée '**console**' et est très liée au fichier de log puisque ce sont les messages passant dans la console qui sont inscrits dans les fichiers de log. Ainsi, une fois que vous avez refermé cette fenêtre de console, ouvrez le fichier de log correspondant : c'est un fichier " **.log* " qui se trouve par défaut dans le répertoire "*c:/program files/SCOL/log*", et dont le nom contient la date (an-mois-jour_heure-minute-seconde). Vous trouverez à la fin de ce fichier la réplique de ce qui est reproduit ici. Bravo, vous avez réalisé votre premier programme en SCOL. Essayons de comprendre ce qui s'est produit.

D'une manière générale, les fichiers **.SCOL* sont les fichiers de démarrage appelés **fichiers script**, alors que les fichiers **.pkg* sont des fichiers contenant des programmes écrits en langage SCOL. Nous verrons plus loin qu'il existe un lien étroit entre les deux types de fichier. Nous détaillerons plus loin la syntaxe des fichiers *script*.

Le fichier *hello1.SCOL* contient les commandes de démarrage du programme. La première ligne (`'_load …'`) indique qu'il faut charger le programme *hello1.pkg*. La seconde ligne indique qu'il faut ensuite exécuter la fonction '*main'*.

Le fichier *hello1.pkg* contient le programme proprement dit. Dans un programme SCOL, on trouve plusieurs sortes de définitions :

- des fonctions, dont la définition commence par '`fun`'
- des variables, dont la définition commence par '`var`' ou '`typeof`'
- des nouveaux types, dont la définition commence par '`typedef`' ou '`struct`'
- et d'autres choses encore...

Chaque définition se termine par un double point-virgule '`;;`'. Ceci est un clin d'œil au langage Caml développé par l'Inria.

Les commentaires sont placés comme en C entre `/*` et `*/`. A la différence du C, on peut imbriquer les commentaires, ce qui est souvent très pratique.

Dans le fichier *hello1.pkg*, on définit donc une fonction appelée '`main`'. Le nom de cette fonction est suivi de `()` : elle ne prend aucun argument. La valeur de la fonction suit le signe `=`. Chaque expression de la fonction est séparée de la suivante par un point-virgule (`';'`). Ici, la fonction comporte donc deux expressions : `_showconsole` et `_fooS ">>>>>>>>>>>> Hello World"`

La définition de la fonction se termine par un double point-virgule.

La première expression appelle la fonction '`_showconsole`' qui a pour effet d'afficher la fenêtre de console (celle-ci reste habituellement cachée).

La seconde expression appelle la '`_fooS`' qui nécessite un argument de type chaîne de caractères, et l'affiche dans la fenêtre de console et donc dans le fichier de log.

En langage SCOL, les arguments ne sont pas placés entre parenthèses ni séparés par des virgules, comme c'est le cas pour le langage C. Les arguments suivent la fonction, simplement séparés par des espaces ou des retours à la ligne. Si l'on souhaite mettre des parenthèses, on peut en mettre soit autour d'un argument, soit autour de l'ensemble fonction+arguments. Par exemple, les trois formulations suivantes sont correctes :

```
_fooS (">>>>>>>>>>>> Hello World")
(_fooS ">>>>>>>>>>>> Hello World")
(_fooS (">>>>>>>>>>>> Hello World"))
```

Nous reviendrons plus loin sur ce point important. Mais notons toutefois que si les parenthèses vous déplaisent, vous pouvez utiliser à la place des accolades : c'est équivalent.

Lorsque la fenêtre de console s'affiche, la machine virtuelle est toujours en fonctionnement. Elle ne s'arrêtera que lorsque vous détruirez la fenêtre de console. C'est en effet un des moyens de détruire une machine SCOL.

On remarquera, vers la fin du fichier de log (4 lignes avant la mention '`hello world`') la ligne suivante :

```
fun main : fun [] S
```

Cette ligne est très importante : elle indique que la machine virtuelle a bien enregistré (compilé) la fonction '`main`', et elle en donne le type : une fonction qui ne prend aucun argument et qui retourne une chaîne de caractères. En effet, en SCOL, vous n'aurez que très rarement à définir vous-même le

type d'une fonction (nombre et type des arguments, type du résultat). La machine SCOL le fait à votre place (on parle "d'inférence de type").

2.2. Seconde version

fichier '*Tutorial/hello2.SCOL*'

```
_load "Tutorial/hello2.pkg"
main "Test"
```

fichier '*Tutorial/hello2.pkg*'

```
/* Hello2.pkg */

fun end(a,b,r)=
 _closemachine;;

fun main(title)=
 _DLGrflmessage
 (_DLGMessageBox _channel nil title "Hello World" 0)
 @end 0;;
```

Cette fois, la fenêtre de console n'apparaît pas, mais à la place une boîte de dialogue toute simple.

Ce programme met plusieurs points en évidence.

Tout d'abord, la fonction '`main`' utilise ici un argument appelé '`title`'. On voit ici comment passer cet argument depuis le fichier *hello2.SCOL*. Il vaudra ici "`Test`".

La fonction '`main`' du fichier *hello2.pkg* n'utilise qu'une seule expression, mais celle-ci appelle une fonction '`_DLGrflmessage`' (qui requiert 3 arguments) dont le premier argument est lui même l'invocation d'une fonction '`_DLGMessageBox`' (qui requiert 5 arguments). Sans entrer dans les détails, disons que la fonction '`_DLGMessageBox`' crée une boîte de dialogue dont le titre est le 3ème argument et le texte est le 4ème. Cette fonction retourne l'identifiant de la boîte de dialogue. La fonction '`_DLGrflmessage`' permet de définir ce qui se passera lorsque l'utilisateur fermera la boite de dialogue. Le premier argument est la boîte de dialogue, le second est la fonction à appeler. Le signe '`@`' permet de passer en quelque sorte un pointeur vers la fonction '`end`' (pour reprendre la terminologie du langage C). Sans ce signe, la fonction '`end`' serait appelée immédiatement (ce qui provoquerait une erreur de compilation puisque la fonction '`end`' nécessite trois arguments, alors qu'il n'y en a ici qu'un seul disponible).

La fonction '`end`' contient une seule expression qui appelle la fonction '`_closemachine`' qui a pour conséquence de fermer la machine SCOL.

Un coup d'oeil dans le fichier de log (*SCOL/hello2.SCOL.log*) révèle la ligne suivante :

```
fun main : fun [S] MessageBox
```

Cette fois, le compilateur a détecté que la fonction '`main`' utilise un argument de type S (chaîne de caractères) et retourne le type `MessageBox`. Il est remarquable que le développeur n'a pas eu à préciser le type de l'argument (`S`), mais que le compilateur l'a lui-même déterminé.

2.3. Troisième Version

fichier '*Tutorial/hello3.SCOL*' :

```
_load "Tutorial/hello3.pkg"
main
```

fichier '*Tutorial/hello3.pkg*' :

```
/* Hello3.pkg */

fun _end(a,b)=_closemachine;;

fun _resize(a,t,x,y)=_SIZEtext t x-2 y-2 1 1;;

fun main()=
 let _CRwindow _channel nil 150 150 400 300
  WN_MENU|WN_MINBOX|WN_SIZEBOX "Hello World"
 -> win in
 let _CRtext _channel win 1 1 398 298
    ET_VSCROLL|ET_HSCROLL "Hello World"
 -> text in
 (_CBwinDestroy win @_end nil;
  _CBwinSize win @_resize text
 );;
```

Ce programme définit trois fonctions : `_end`, `_resize` et `main`.

Nous voyons apparaître ici plusieurs éléments nouveaux : `_SIZEtext`, `let`, `nil`, `_CRwindow`, `_CRtext`, `_CBwinDestroy` et `_CBwinSize`.

Les fonctions `_SIZEtext`, `_Crwindow`, `_Crtext`, `_CbwinDestroy` et `_CbwinSize` sont des fonctions de l'interface graphique :

`_CRwindow` permet de créer une fenêtre :

- l'argument 1 est le canal propriétaire de la fenêtre (ceci sera expliqué plus loin),
- l'argument 2 est la fenêtre mère de la fenêtre à créer,
- les arguments 3 et 4 donnent la position de la fenêtre,
- les arguments 5 et 6 donnent la taille de la fenêtre,
- l'argument 7 contient les flags de la fenêtre (ici, on définit une fenêtre simple, que l'on peut minimiser et dont on peut changer la taille à la souris),
- l'argument 8 contient le titre de la fenêtre.

`_CRtext` permet de créer une zone de texte :

- l'argument 1 est le canal propriétaire de la zone de texte (ceci sera expliqué plus loin),
- l'argument 2 est la fenêtre mère de la zone de texte à créer,
- les arguments 3 et 4 donnent la position de la zone de texte,
- les arguments 5 et 6 donnent la taille de la zone de texte,
- l'argument 7 contient les flags de la zone de texte (ici, on définit une zone de texte avec ascenseurs horizontal et vertical),
- l'argument 8 contient le texte placé initialement dans la zone de texte.

`_CBwinDestroy` définit la fonction à appeler lorsque la fenêtre est détruite :

- l'argument 1 est la fenêtre en question,
- l'argument 2 est la fonction à appeler lors de la destruction,
- l'argument 3 est un paramètre utilisateur.

`_CBwinSize` définit la fonction à appeler lorsque la fenêtre est redimensionnée :

- l'argument 1 est la fenêtre en question,
- l'argument 2 est la fonction à appeler lors du redimensionnement,

- l'argument 3 est un paramètre utilisateur.

`_SIZEtext` redimensionne une zone de texte :

- l'argument 1 est la zone de texte à redimensionner,
- les arguments 2 et 3 donnent les nouvelles dimensions de la zone,
- les arguments 4 et 5 donnent les nouvelles coordonnées de la zone.

On remarque la fonction `'nil'`. En fait `nil` est une valeur particulière, qui n'a pas de type défini. Toute variable, tout paramètre peut valoir `nil`. Ici, `nil` est utilisé à deux endroits :

- comme fenêtre mère de la fenêtre créée par `'_Crwindow'` : cela signifie que la fenêtre n'a pas de mère
- comme paramètre utilisateur de la fonction `'_CbwinDestroy'` : cela peut signifier qu'on n'a pas de paramètre utilisateur à passer.

La fonction `'let ... -> ... in ...'` permet de définir des variables locales.

En effet, entre `'let'` et `'->'`, on écrit une expression quelconque. Comme toute expression en SCOL, elle retourne un résultat (ici, c'est une fenêtre dans le premier cas, une zone texte dans le deuxième).

Entre `'->'` et `'in'`, on écrit la variable locale (ici, `'win'` et `'text'`).

Puis après le `'in'`, on écrit l'expression dans laquelle la variable locale est utilisable. Au-delà de l'expression qui suit le `'in'`, la variable locale n'est plus reconnue.

Un mot sur les fonctions callback : les fonctions passées en argument des fonctions `'_CbwinDestroy'` et c sont des fonctions dites de callback (ou de réflexe, en français), car elles seront appelées ultérieurement, lorsqu'un certain événement se produira. Par convention, les fonctions de callback utilisent au moins deux arguments :

- le premier argument redonne l'objet affecté (la fenêtre dans le cas de `'_CbwinDestroy'`, et `'_CbwinSize'`)
- le deuxième argument est le paramètre utilisateur tel qu'il a été défini par la fonction `'_CbwinDestroy'` ou `'_CbwinSize'`.

Les arguments supplémentaires éventuels dépendent de la nature de l'événement. Par exemple la callback de destruction n'utilise pas de paramètre supplémentaire, tandis que la callback de redimensionnement utilise deux paramètres supplémentaires qui contiennent la nouvelle taille de la fenêtre.

Nous sommes maintenant prêts à comprendre ce troisième programme.

La fonction `'main'` crée une fenêtre ainsi qu'une zone de texte incluse dans cette fenêtre. Le titre de la fenêtre ainsi que le contenu de la zone texte sont positionnés sur la valeur `'Hello World'`. La fonction `'main'` définit également deux callbacks sur la destruction et le redimensionnement de la fenêtre.

En cas de destruction de la fenêtre, le programme s'arrête (fonction `'_closemachine'`).

En cas de redimensionnement de la fenêtre, il faut redimensionner la zone *texte*. Ceci est fait en appelant la fonction `'_SIZEtext'`. On remarquera l'utilisation du paramètre utilisateur pour 'passer' la zone de texte à la callback.

Ce petit programme va nous servir dans le chapitre suivant, puisqu'il permet de définir une zone d'affichage simple.

3. PRINCIPES DE LA PROGRAMMATION SCOL

3.1. Les environnements et les canaux

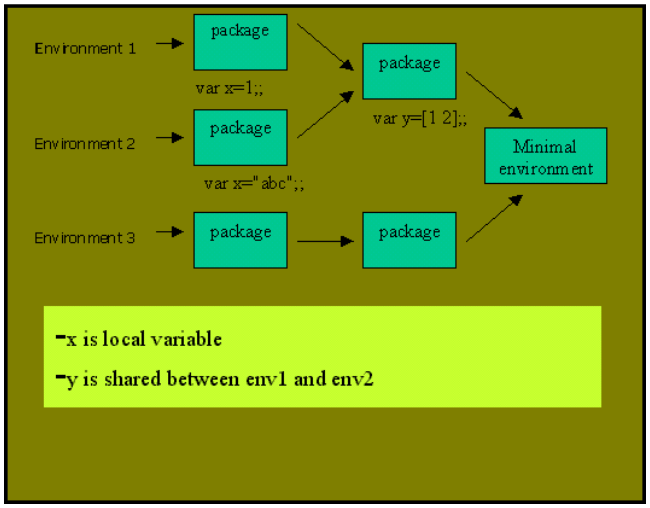
3.1.1. Environnements

Un **environnement** est une liste (au sens informatique) de variables et de fonctions. Dans l'exemple '*hello3*' du chapitre précédent, il y avait dans la machine SCOL un environnement contenant les fonctions `'main'`, `'_resize'`, `'_end'`, ainsi que toutes les variables définies dans le fichier *locked/lib/const*. De plus cet environnement contenait toutes les fonctions des API SCOL (`'_Crwindow'`, `'_Crtext'`, ...).

Lorsqu'on compile un fichier écrit en SCOL, cela se fait toujours dans un certain environnement. Le programme ainsi compilé peut faire référence aux fonctions et variables de l'environnement, et les fonctions et variables définies dans le fichier s'ajoutent à l'environnement, c'est-à-dire au **début** de la liste de variables de fonctions : en compilant le fichier '*hello3.pkg*', on a ajouté en tête de l'environnement les fonctions `'main'`, `'_resize'` et `'_end'`. Ceci signifie que tout fichier compilé **après** pourrait référencer ces trois fonctions.

Par définition, on appellera **environnement minimal** l'environnement qui ne contient que l'API SCOL.

L'originalité de la machine SCOL est de pouvoir gérer plusieurs environnements en même temps : plusieurs listes de variables et de fonctions coexistent dans la mémoire de la machine. C'est pourquoi il est important de savoir, à tout moment, dans quel environnement on travaille. Ces environnements ne sont pas forcément indépendants. Comme on l'a annoncé, un environnement est une liste. Il est possible d'avoir deux listes ayant une partie finale identique. Par exemple, les listes (1,2,10,11,12) et (3,4,5,10,11,12) ont une fin commune de taille 3 : (10,11,12). De même, en SCOL, deux environnements peuvent avoir une fin commune. De fait, chaque environnement se termine par l'**environnement minimal**. Ceci permet de partager certaines ressources entre les environnements : fonctions et variables peuvent être mises en commun. On appellera variable **locale à un environnement** une variable qui est définie dans une partie de la liste non partagée de l'environnement.



3.1.2. Canaux

Un **canal** est un couple (environnement, liaison réseau). Cette association est une originalité de SCOL. L'environnement est au minimum l'environnement minimal. La liaison réseau est normalement une liaison Tcp-Ip de type '*socket*'. Cependant, on définit un type de canal particulier, dit **unplugged**, qui ne contient pas de liaison réseau.

Lorsque l'on crée un canal, il faut donc spécifier l'environnement initial de ce canal ainsi que l'éventuelle liaison réseau. L'environnement d'un canal peut évoluer au cours du temps :

- il est possible d'agrandir un environnement en compilant de nouveaux fichiers de langage SCOL,
- il est possible de supprimer des éléments d'un environnement en retranchant les fonctions et variables situées en tête d'un environnement,
- il est possible de substituer un environnement par un autre, et notamment de réinitialiser un canal en y replaçant l'environnement minimal.

3.1.3. Démarrage de la machine SCOL : premiers éléments

Au démarrage de la machine SCOL, un canal *unplugged* est automatiquement créé, avec l'environnement minimal. Le fichier **.SCOL* permet de définir les opérations à effectuer sur ce canal : typiquement, il s'agit de compiler un ou plusieurs fichiers de langage SCOL, puis de lancer l'exécution d'une fonction. Les fichiers seront compilés dans ce premier canal, et la fonction sera recherchée dans l'environnement de ce canal.

3.2. Programmation fonctionnelle

Le langage SCOL est un langage d'inspiration **fonctionnelle**, même s'il autorise une programmation impérative, ainsi que tous les effets de bords.

L'unité de base du langage SCOL est donc la fonction : une fonction est un objet informatique qui consomme un certain nombre d'arguments (éventuellement réduit à zéro), et qui produit un et un seul résultat.

Toute fonction retourne donc un résultat, même lorsque ce résultat n'est pas "significatif". Par exemple, dans l'exemple *hello1*, la fonction `'_showconsole'` retourne un résultat (en fait un entier). Pourtant, ce résultat n'est pas "important", car ce qui compte pour le développeur, c'est le fait que la fonction `'_showconsole'` provoque l'affichage de la fenêtre console. En fait, tout effet d'une fonction autre que le résultat de la fonction est appelé **effet de bord**. Un effet de bord ne doit pas faire oublier que la fonction qui l'a produit a elle-même retourné un résultat.

Par son approche fonctionnelle, SCOL incite à l'imbrication de fonctions. Nous avons déjà vu un exemple d'imbrication dans le programme *hello2*. En voici un second, et bien d'autres suivront.

```
fun f(x)= x+1;;
fun g(x)= x*2;;
fun gof(x)= g f x;;
```

Ici la fonction `'f'` calcule le suivant `(x+1)`, la fonction `'g'` calcule le double `(x*2)`, la fonction `'gof'` calcule la composée de `'g'` et de `'f' : (x+1)*2` .

Si l'on souhaite mettre des parenthèses par soucis de lisibilité, on pourra écrire :

```
fun gof(x)= g (f x);;
fun gof(x)= (g f x);;
fun gof(x)= (g (f (x)));;
```

Mais on ne pourra pas écrire :
```
fun gof(x)= (g f) x;;
```

Autre exemple :
```
fun f(a,b)=strcat a b;;
fun g(a)=strcat a ".";;
fun h(a,b)= f g a b;;
```

La fonction '`strcat`' est une fonction qui concatène deux chaînes de caractères. Par exemple (`strcat "a" "b"`) vaut `"ab"`.

Dans cet exemple, la fonction '`h`' prend deux chaînes de caractères et retourne une chaîne composée des deux arguments séparés par un point.

On pourrait écrire :
```
fun h(a,b)= f (g a) b;;
fun h(a,b)= (f (g a) b);;
```

Mais on ne pourra pas écrire :
```
fun h(a,b)= f((g a) b);;
```

Et encore moins :
```
fun h(a,b)= f(g(a),b);;
```

Ainsi, les parenthèses entourent les expressions, et non pas, comme en C, la liste des arguments d'une fonction. Chaque argument étant lui-même une expression, on peut entourer un argument de parenthèses. Vous pouvez remplacer les parenthèses par des accolades : c'est parfaitement équivalent.

3.3. Types et typage

3.3.1. Introduction sur les types

Comme il a été indiqué dans les exemples "*Hello World*", les types des fonctions sont déterminés automatiquement par la machine SCOL, lors de la compilation. Cependant, il est certains cas où le développeur devra définir lui-même certains types : définition de structures, de constructeurs de types, de certaines variables et parfois de prototypes de fonctions. De plus, pour pouvoir lire les documentations sur les API SCOL, il est nécessaire de comprendre les types.

Le **type** d'un élément de votre programme peut être très divers : entier, chaîne de caractères, tableau, tuple, fonction, ... En SCOL, les types sont simplement une aide pour le développeur : les types ne sont utilisés que lors de la compilation, et permettent de détecter la plupart des erreurs dès cet instant. Ensuite, lors de l'exécution de votre programme, ils ne sont plus utilisés, car le programme est déjà prouvé comme correctement typé.

Le **typage** est l'opération de vérification des types de votre programme. Il est effectué lors de la compilation (on dit qu'il est **statique**). Par exemple, il détecte que vous utilisez une chaîne de caractères avec une fonction qui nécessite un entier, mais cela peut être beaucoup plus subtil. Lorsqu'une erreur de type est détectée, la machine SCOL s'arrête et produit sur la fenêtre console (et donc dans le fichier de log) un message d'erreur expliquant la nature et la position de l'erreur de type.

En SCOL, le typage se fait par **inférence de type** : cela signifie que le compilateur calcule lui-même le type de vos fonctions, vous n'avez normalement pas à le préciser vous-même. Le typage SCOL est également dit **polymorphe**. En effet, certaines fonctions n'imposent pas de condition de type sur certains arguments, ou certaines parties de certains arguments.

Par exemple la fonction suivante :
```
fun f(x)=1;;
```

Cette fonction ne contraint pas le type de `x`, puisque cet argument n'est pas utilisé. La fonction `'f'` est dite polymorphe car elle accepte des arguments de types différents.

Cet exemple est particulièrement trivial, mais il y a des exemples plus intéressants : une fonction qui calcule la taille d'une liste ne se soucie normalement pas du type des éléments de la liste. Plutôt que d'écrire une fonction pour les listes d'entiers, une autre pour les listes de chaînes de caractères, ..., le polymorphisme vous permet de n'écrire qu'une seule fonction.

3.3.2. Syntaxe des types SCOL

Le langage SCOL définit une syntaxe pour écrire les types. Celle-ci est définie par le tableau suivant :

Type	=	*B*	|	**u***n*	|	**r***n*
	|	**tab***Type*	|	**[***Type**]**	|	**fun [***Type**] *Type***
TypeMono	=	*B*	|	**r***n*		
	|	**tab***TypeMono*	|	**[***TypeMono**]**	|	**fun [***TypeMono**] *TypeMono***
B	=	Type de base				
u*n*	=	variable liée				
r*n*	=	récursion de niveau *n*				

Les types de base sont :

I	:	int
S	:	string
F	:	float
Chn	:	canal SCOL
Srv	:	serveur SCOL
Env	:	environnement
Comm	:	communication

Cette liste n'est pas exhaustive : grâce aux structures et aux constructeurs de types, vous pouvez vous-même développer vos types de base.

Quelques commentaires sur le tableau, si vous n'êtes pas familier avec ces notations.

La première ligne définit l'expression *Type*, qui est en fait le type SCOL. Puis, séparés par des '|', on trouve les différentes manières d'écrire l'expression : cela peut être :

- `B`, qui est défini à la troisième ligne : c'est un type de base, tel que `I, S, F`, ... Donc puisque `I` est un type de base, `B` peut s'écrire `I`, et puisque `Type` peut s'écrire `B`, `I` est bien un type en SCOL.
- **u***n* avec *n* entier : `u0, u1, u2, u3`, ... sont des types : ils correspondent aux variables liées (nous préciserons plus loin).
- **r***n* avec `n` entier : `r0, r1, r2, r3`, ... sont des types : ils définissent les récursions dans les types (nous préciserons plus loin)
- **`tab`** *`Type`* : type tableau. Le mot **tab** est suivi du type des éléments du tableau. Par exemple `tab I` est le type d'un tableau d'entiers.
- `[`*`Type`*`*]` : type tuple. Entre crochets, on écrit plusieurs `Types` (c'est le sens de l'étoile *). Par exemple, `[I S]` est un tuple de deux éléments dont le premier est un entier et le second une chaîne de caractères. Eventuellement, le tuple est vide : `[ ]`. Le tuple peut lui-même contenir des tuples : `[I [S I]]`

- **fun** `[`*Type**`]` `Type` : type fonction. le mot `fun` est suivi d'un tuple contenant les arguments de la fonction puis du type du résultat. Par exemple `'fun [I I] S'` est une fonction qui prend deux entiers en arguments, et retourne une chaîne de caractères.

L'expression `TypeMono` définit les types monomorphes (non polymorphes) : la seule différence avec *Type* est l'absence des variables liées **u***n*.

3.3.3. Quelques précisions sur les types

3.3.3.1. Polymorphisme

Reprenons l'exemple de polymorphisme `fun f(x)=0;;` Si vous compilez un exemple contenant cette fonction, vous verrez dans le fichier de log que le type détecté par le compilateur est : `fun [u0] I.`

`u0` est mis pour 'Unknown 0'. Cela signifie que le type n'a pas pu être déterminé et qu'il est indifférent. Le `0` permet de différencier les types inconnus.

Autre exemple : `fun f(x)=x;;`

Cette fois le compilateur va déterminer le type suivant : `fun [u0] u0.`

Cela signifie que le type de l'argument `x` est indifférent, mais que le type du résultat est le même.

Autre exemple : `fun f(x,y)=x;;`

Cette fois le compilateur va déterminer le type suivant : `fun [u0 u1] u0.`

Cela signifie que le type des arguments est indifférent, mais que le type du résultat est celui du premier argument.

3.3.3.2. Récursion

Nous avons vu que les tuples peuvent être imbriqués `[I [S I]]`. Que ce passe-t-il lors d'une imbrication infinie ?

Introduisons la notion de liste en SCOL. Le type *List* n'existe pas en SCOL, par convention, on définit les listes comme un tuple de deux éléments dont le premier est le premier élément de la liste, et le second la suite de la liste. On note `nil` la fin de la liste.

En SCOL, on écrira la liste des entiers de 1 à 5 de la manière suivante : `1::2::3::4::5::nil`, ou encore, de manière absolument équivalente : `[1[2[3[4[5 nil]]]]]`.

Le type d'une liste d'entiers devrait être : `[I[I[I[I ...]]]]`

On notera `[I r1]` le type d'une telle liste : tuple dont le deuxième élément est une récursion de niveau 1.

Autre exemple : soit une 'liste' alternée : `[I[S[I[S ...]]]]`. Cette 'liste' alterne des entiers et des chaînes de caractères. On notera `[I[S r2]]` le type d'une telle liste : tuple dont le deuxième élément est un tuple dont le deuxième élément est une récursion de niveau 2. Note : ceci n'est pas une vraie liste, car une vraie liste ne contient normalement que des éléments de même type.

D'une manière générale, les seules types récursifs dont vous aurez vraisemblablement à vous préoccuper seront les listes, et donc `[I r1]` pour une liste d'entiers, `[S r1]` pour une liste de chaînes de caractères,... Cependant, il est possible que vous réalisiez par erreur ou non des fonctions dont le type contient des éléments de récursion différents.

Pour ceux qui souhaitent aller plus loin :

Il est possible de représenter un type par un graphe orienté. Les nœuds sont :

- les types de bases (ce sont alors des feuilles),
- les types polymorphes (ce sont alors des feuilles),
- les tableaux : il y a alors un fils qui représente le type des éléments du tableau,
- les *n*-tuples : il y a alors *n* fils correspondant chacun à un des éléments du tuple,
- les fonctions : il y alors 2 fils correspondant le premier au tuple des arguments, le second au résultat.

Ce graphe ressemble en fait plus à un arbre. Mais de temps à autres, une branche peut partir d'un nœud vers un nœud qui est : lui-même, son père, son grand-père, ou plus encore. C'est ici qu'intervient la récursion : r1 pour une branche allant vers lui-même, r2 pour une branche allant vers son père, ...

3.3.3.3. Quelques contraintes

La constante `nil` n'a pas de type défini, ou plutôt elle a tous les types à la fois.

Un type n'est valide que s'il ne contient pas de variable libre : "`u0`" et "`fun [ ] u0`" sont syntaxiquement corrects, mais sont invalides.

Typiquement, si vous écrivez une fonction dont le type retourne un résultat contenant un u*n* (par exemple `fun [ ] u0`), cela signifie que votre fonction retourne `nil` dans tous les cas. Remplacez alors simplement un des `nil` par 0.

Il y a une restriction au polymorphisme due aux effets de bords : les variables sont nécessairement monomorphes, alors que les fonctions peuvent être polymorphes.

3.4. Syntaxe du langage SCOL

Comme tout langage de programmation, SCOL utilise une syntaxe bien précise. Le tableau suivant la définit complètement. Si vous n'êtes pas familier avec ce type de notation, reportez vous au chapitre précédent sur les types : la syntaxe des types utilise la même notation, mais est plus simple. Rappelons simplement que les `*` indiquent que l'élément est répété un certain nombre de fois (éventuellement zéro). Les accolades laissent le choix entre plusieurs éléments. Par exemple, *{***I**,**S***}** correspond à une suite quelconque de **I** et de **S**. Les caractères en gras correspondent aux éléments de syntaxe que l'on retrouve dans le fichier source d'un programme SCOL, tandis que les éléments en italiques sont des éléments de réécriture dont la signification est donnée ailleurs dans le tableau.

Un fichier écrit en langage SCOL contient simplement l'élément *SCOL* tel qu'il est défini dans le tableau. L'élément *SCOL* est simplement une suite de définitions (*Definition* dans le tableau).

Les définitions sont de 8 sortes :

- `fun` : définition de fonction
- `typeof` : définition d'une variable par son type
- `var` : définition et initialisation d'une variable
- `struct` : définition d'un type de structure
- `typedef` : définition de constructeurs de types
- `defcom` : définition d'un constructeur de communication
- `defcomvar` : définition d'un constructeur de communication variable
- `proto` : définition d'un prototype de fonction

Vous pouvez sauter le tableau suivant, mais vous le trouverez sans doute bien pratique plus tard.

SCOL = *Definition**
Definition = **fun** *Function* **(***Args***) =** *Program* **;;**
| **typeof** *Var* **=***TypeMono* **;;**
| **var** *Var* **=** *Val* **;;**
| **struct** *NewType* **= [** *Fields* **]** *Function* **;;**
| **typedef** *NewType* **=** *TypeConstr* **;;**
| **defcom** *Com* **=** string *{***I**,**S***}** **;;**
| **defcomvar** *Comvar* **=** *{***I**,**S***}** **;;**
| **proto** *Function* **=** *Type* **;;**

Program	=	*Expr*	|	*Expr* **;** *Program*		
Expr	=	*Arithm*	|	*Arithm* **::** *Expr*		
Arithm	=	A_1	|	A_1 **&&** *Arithm*	|	A_1 **||** *Arithm*
A_1	=	A_2	|	**!**A_1		
A_2	=	A_3	|	A_3 **==** A_3	|	A_3 **!=** A_3
	|	A_3 **<** A_3	|	A_3 **>** A_3	|	A_3 **<=** A_3
	|	A_3 **>=** A_3	|	A_3 **=.** A_3	|	A_3 **!=.** A_3
	|	A_3 **<.** A_3	|	A_3 **>.** A_3	|	A_3 **<=.** A_3
	|	A_3 **>=.** A_3				
A_3	=	A_4	|	A_4 **+** A_3	|	A_4 **-** A_3
	|	A_4 **+.** A_3	|	A_4 **-.** A_3		
A_4	=	A_5	|	A_5 ***** A_4	|	A_5 **/** A_4
	|	A_5 ***.** A_4	|	A_5 **/.** A_4		
A_5	=	A_6	|	A_6 **&** A_5	|	A_6 **|** A_5
	|	A_6 **^**A_5	|	A_6 **<<** A_5	|	A_6 **>>** A_5
A_6	=	*Term*	|	**-**A_6	|	**~**A_6
Term	=	**(** *Program* **)**	|	**(** *Program* **;)**		
	|	**{** *Program* **}**	|	**{** *Program* **;}**		
	|	int	|	'char	|	`nil`
	|	string	|	**[***Arithm** **]**		
	|	*Var*(**.***Term*)*	|	**set** *Var*(**.***Term*)* **=** *Arithm*		
	|	*Var*(**.***NameOfField*)*	|	**set** *Var*(**.***NameOfField*)* **=** *Arithm*		
	|	*Function* $Args_{Function}$	|	**@***Function*		
	|			**let** *Arithm* **->** *Locals* **in** *Arithm*		
	|			**if** *Arithm* **then** *Arithm* **else** *Arithm*		
	|			**while** *Arithm* **do** *Arithm*		
	|			**mutate** *Arithm* **<-[** *{_ , Arithm}** **]**		
	|			**exec** *Arithm* **with** *Arithm*		
	|	*Constr Arithm*	|	*Constr0*	|	**match** *Arithm* **with** *Case*

$Args_F$	=	*Arithm* ...*Arithm* : autant de fois *Arithm* que la fonction F a d'arguments
Args	=	nothing | *Args'*
Args'	=	*Local* | *Local* **,** *Args'*
Locals	=	*Local* | **[** *Locals'***]**
Locals'	=	*{_ , Locals}**
Val	=	*Val'* | *Val'* **::** *Val*

Val' = int | 'char | **nil**
| string | - int | **[** *Val** **]**
| **(***Val***)**

Fields = *Field* | *Field*, *Fields*
Field =*NameOfField* **:** *TypeMono*

TypeConstr = *TypeConstr'* |*TypeConstr'* **|** *TypeConstr*
TypeConstr'= *Constr TypeMono* | *Constr0*

Case = *Case'* | *Case'* **|** *Case* | **(** **_** **->** *Arithm***)**
Case' =**(** *Constr Local* **->** *Arithm* **)** | **(***Constr0* **->** *Arithm* **)**

- `Var` = nom de variable
- `Function` = nom de fonction
- `NewType` = nouveau type de base défini par le développeur
- `Local` = variable locale (liée)
- `NameOfField` = nom de champ dans une structure
- `Constr` = constructeur de type
- `Constr0` = constructeur de type vide
- `Com` = constructeur de communication
- `Comvar` = constructeur de communication variable
- `int` = entier
- `char` = caractère
- `string` = chaîne

Les entiers peuvent être codés dans les bases suivantes :

- en décimal : 12349
- en hexa : 0x3fe
- en binaire : 0b10011
- en octal : 0o234235

Ils sont codés sur 31 bits signés.

Les `char` permettent de récupérer le code ascii d'un caractère : '`A` est un entier qui vaut 65.

Les chaînes de caractères sont entre guillemets. Le caractère `\` permet d'accéder à certaines commandes :

- `\n` : retour chariot
- `\z` : caractère NULL
- `\"` : guillemet
- `\\` : \
- `\nombre en décimal` : \132 est le caractère Ascii 132

Un `\` en fin de ligne permet de signaler au compilateur de ne pas tenir compte du retour à la ligne.

Les remarques sont, comme en C, entre `/*...*/` et peuvent être imbriquées les unes dans les autres.

Les constructions `(Program)`, `(Program ;)`, `{Program}` et `{Program ;}` sont équivalentes.

3.5. Constructions de base

3.5.1. Eléments principaux

3.5.1.1. Variables

Vous pouvez définir une variable en utilisant `typeof`. Par exemple :

```
typeof x=[I I];;
```

La variable `x` est créée, c'est un tuple de deux entiers. La variable est initialisée à `nil`.

Si vous souhaitez lui attribuer directement une valeur, vous utiliserez `var` :

```
var x=[1 2];;
```

Cependant, `var` ne permet pas certaines opérations, notamment les opérations arithmétiques.

Vous pouvez modifier la valeur d'une variable en utilisant la construction : **set** *Var* **=** *Arithm*

Par exemple : `set x=a+b`

Ceci constitue un effet de bord : ce qui vous intéresse n'est pas le résultat de la fonction '`set`', mais bien le fait que la valeur de `x` a changé.

3.5.1.2. Fonctions

Pour définir une fonction, placez `fun` avant le nom de la fonction et faites suivre par la liste des arguments entre parenthèses et séparés par des virgules. Puis le signe '`=`' précède le corps de la fonction qui se termine par un double point-virgule. Exemple, la fonction '`somme`' :

```
fun sum(x,y)=x+y;;
```

Le typage va déterminer le type de la fonction : `fun [I I] I`

La fonction prend effectivement deux entiers et retourne un entier.

Vous pouvez écrire les expressions arithmétiques de manière classique : `(x+y)*z/w`. Vous disposez également des opérateur logiques du C : `&, |, ^, <<, >>, ~, &&, ||, !, ==, !=, <, >, <=, >=.`

Les opérations arithmétiques ainsi que les opérations de comparaisons sont également disponibles pour les nombres flottants (type F), il faut cependant faire suivre l'opérateur d'un point : `+., -., >., ...`

3.5.1.3. Tests et conditions

Une des constructions les plus importantes en programmation est la construction conditionnelle, le traditonnel '`if ... then ... else ...`'. Evidemment, cette construction existe en SCOL, sous la même forme :

```
if C then T else F
```

où `C` est une condition, `T` (respectivement `F`) est l'expression à réaliser si la condition est vraie (respectivement fausse).

En SCOL, on doit toujours définir l'expression '`else`'.

La condition est nécessairement une expression retournant un entier. Le résultat de la condition est considéré comme " vrai " si l'entier est différent de zéro, et faux s'il vaut 0. Dans l'expression condition, on peut utiliser les opérateurs booléens du C : `&&` et `||`. Comme en C, l'expression n'est pas forcément évaluée complètement :

- `A && B` : si A est faux, B n'est pas évalué (le résultat est forcément faux)
- `A || B` : si A est vrai, B n'est pas évalué (le résultat est forcément vrai)

Les expressions `T` et `F` doivent retourner le même type, qui est le type retourné par la construction.

En effet, la construction '`if ... then ... else ...`' est une fonction qui retourne la valeur de `T` ou de F selon la valeur de la condition. Le type de cette fonction est :

```
fun [I u0 u0] u0
```

On peut donc intégrer la construction dans une expression :

```
1+if x==0 then 1 else 3
```

Cette expression vaudra 2 si `x` vaut 0, 4 sinon.

De même la fonction suivante :

```
fun f(x)=if x then "a" else "b";;
```

La fonction '`f`' retourne la chaîne "`a`" si `x` n'est pas nul, "`b`" sinon. Dans les langages impératifs tels que le C, on n'a pas ce type de fonctionnement.

3.5.1.4. Exemple

Pour vous permettre d'effectuer vos tests, nous modifions le programme *hello3* présenté dans un chapitre précédent.

fichier '*Tutorial/mytest.SCOL*' :

```
_load "Tutorial/mytest.pkg"
main
```

fichier '*Tutorial/mytest.pkg*' :

```
/* MyTest.pkg */

/* My tests */
fun sum(x,y)=x+y;;

fun mymain()=
 itoa sum 10 30;;

/* Common part */

fun _end(a,b)=_closemachine;;

fun _resize(a,t,x,y)=_SIZEtext t x-2 y-2 1 1;;

fun main()=
 let _CRwindow _channel nil 150 150 400 300
  WN_MENU|WN_MINBOX|WN_SIZEBOX "My Test"
```

```
-> win in
let _CRtext _channel win 1 1 398 298
   ET_VSCROLL|ET_HSCROLL ""
-> text in
(_CBwinDestroy win @_end nil;
 _CBwinSize win @_resize text;
 _SETtext text mymain
);;
```

Dans cet exemple, on affiche simplement une fenêtre texte dans laquelle on écrit (fonction `_SETtext`) le résultat de la fonction `mymain`.

La fonction `mymain` doit toujours retourner une chaîne de caractères. Si vous manipulez des entiers, utilisez la fonction `itoa` pour les convertir en chaîne de caractères avant de les retourner. A l'aide de `itoa` et de `strcat` (qui concatène deux chaînes de caractères), vous êtes capables d'afficher tous les résultats.

Ici nous testons la fonction `sum` sur deux entiers `10` et `30`.

Dans les exemples suivants, nous donnerons simplement les fonctions qu'il faudra mettre à la place de `mymain` et `sum`. Vous pouvez faire vous même vos essais.

3.5.2. Utilisation des tuples

Les tuples sont une manière très pratique de manipuler des ensembles hétérogènes de données. En fait, leur utilisation masque une allocation automatique de mémoire, c'est pourquoi les tuples n'existent pas en C. Seul un langage qui gère automatiquement allocation et désallocation de la mémoire peut réellement utiliser les tuples.

Le terme [*Arithm**] construit un tuple à partir des différentes expressions présentes entre les crochets. Par exemple:

```
[ 1 2  nil [ 3 4 ] ]
```

est un tuple de taille 4, dont le dernier élément est un tuple de taille 2.

On récupère les composantes du tuple grâce à la fonction `let` :

```
let [1 2 nil [ 3 4 ] ] -> [ a _ b c ] in A
```

dans l'expression `A`, les variables `a`, `b` et `c` valent respectivement `1`, `nil` et `[ 3 4 ]`. On 'saute' une valeur du tuple en mettant un caractère underscore '_' ; cela signifie qu'on ne souhaite pas utiliser cette valeur et donc qu'on ne prend pas la peine de définir une variable locale pour la recevoir.

On modifie un ou plusieurs champs d'un tuple grâce à la fonction `mutate` :

```
let [1 2 nil [ 3 4 ] ] -> tupletest in
mutate tupletest <- [ _ 5 6 _ ]
```

Dans cet exemple, on commence par créer un tuple appelé `tupletest`. La fonction `mutate` remplace alors les valeurs 2 et `nil` du tuple par respectivement 5 et 6, sans toucher aux autres champs (là où on a écrit un underscore '_'). Cette fonction est cependant dangereuse puisqu'elle modifie en même temps toutes les variables qui pointent directement ou indirectement vers le tuple. Il est plus propre de recréer un tuple.

Exemple : nous allons créer un tuple de taille 3, modifier le deuxième élément avec `mutate`, et afficher son état avant et après la modification.

```
/* My tests */
fun Tuple3toStr(t)= let t->[a b c] in
 strcat strcat strcat strcat strcat strcat
```

```
  "[" itoa a " " itoa b " " itoa c "]\n";;

fun mymain()=
 let [1 2 3]-> mytuple in
 strcat Tuple3toStr mytuple
 (mutate mytuple <- [_ 10 _];
  Tuple3toStr mytuple);;
```

On remarque ici que la valeur de l'expression **(**`Arithm1` **;** `Arithm2`**)** est la valeur du dernier terme `Arithm2`. Ceci signifie que le résultat du premier terme `Arithm1` est perdu. Cette approche impérative (non fonctionnelle) est habituellement due à un effet de bord : ici, c'est la fonction `'mutate'` qui crée l'effet de bord. On peut noter au passage l'inconvénient d'une telle démarche : le résultat du premier terme est perdu, ce qui signifie qu'une information a été perdue. Cette information n'a donc pas été vérifiée lors du typage, ce qui fragilise le programme.

Lorsqu'on exécute ce programme, on obtient le résultat suivant sur la fenêtre texte :

```
[1 2 3]
[1 10 3]
```

3.5.3. *Application aux listes*

Comme il a été mentionné précédemment, les listes en SCOL sont gérées sous forme de tuples de taille 2. Le premier élément d'un tuple est le premier élément de la liste, le second élément du tuple est la suite de la liste. Par exemple, soit `l` une liste :

```
let l->[val next] in ...
```

Cette expression permet de récupérer dans les variables locales `val` et `next`, respectivement le premier élément de la liste et la suite de la liste. On appelle souvent `'hd'` la fonction qui retourne le premier élément de la liste, et `'tl'` la fonction qui retourne la suite de la liste. Ces fonctions sont présentes dans le langage SCOL, cependant on peut les réécrire en SCOL de la manière suivante :

```
fun hd(l)= let l->[val _] in val;;
fun tl(l)= let l->[_ next] in next;;
```

Le type de ces fonctions est :

```
hd : fun [[u0 r1]] u0
tl : fun [[u0 r1]] [u0 r1]
```

On remarque ici que les deux fonctions sont polymorphes : elles sont utilisables pour n'importe quelle liste.

La liste se termine toujours par la liste vide qui vaut `nil`.

Pour construire une liste, le plus simple est d'utiliser le constructeur de liste `'::'`. Par exemple, l'expression `1::2::3::4::5::nil` construit une liste des cinq premiers entiers. Le type de la fonction `'::'` est :

```
fun [u0 [u0 r1]] [u0 r1]
```

Les listes sont un élément très important dans les langages fonctionnels car ce sont des structures de données de taille illimitée (la gestion automatique de la mémoire facilite leur utilisation), et ce sont des structures de données qui s'accommodent bien d'un traitement récursif.

Attardons-nous un instant sur les listes et étudions les exemples suivants :

3.5.3.1. Taille d'une liste

Dans l'exemple suivant, la fonction `'mysizelist'` calcule la taille d'une liste. Cette fonction est récursive :

- la liste `nil` a pour taille 0.

- toute liste non vide `a` pour taille 1+la taille de la liste privée de son premier élément

La fonction '`mymain`' applique la fonction '`mysizelist`' sur une liste de 5 éléments.

Le programme affiche le nombre 5 dans la fenêtre texte.

```
/* My tests */
fun mysizelist(l)=
 if l==nil then 0
 else let l -> [_ next] in 1 + mysizelist next;;

fun mymain()=
 itoa mysizelist 1::2::3::4::5::nil;;
```

Le type de la fonction '`mysizelist`' est naturellement polymorphe :

```
mysizelist : fun [[u0 r1]] I
```

En fait, la fonction '`sizelist`' est déjà définie dans le langage : vous n'avez donc pas à la réécrire vous-même, mais c'est un bon exemple.

3.5.3.2. Quicksort sur les entiers

Un exemple classique : **Quicksort**.

Pour cela on définit trois fonctions, plus une quatrième qui sert à afficher le résultat.

La fonction '`conc`' prend deux listes `p` et `q` et retourne une nouvelle liste qui concatène dans cette ordre les deux listes. C'est une fonction récursive :

- si `p` est la liste vide, alors la concaténation de `p` et `q` vaut simplement `q`
- sinon la concaténation des listes `p` et `q` est une liste dont :
  - le premier élément est le premier élément de la liste `p`
  - la suite de la liste est la concaténation de la liste `p` privée de son premier élément et de la liste `q` (récursion).

La fonction '`dividelist`' est un peu plus compliquée : son rôle est de diviser une liste en deux sous-listes en fonction d'un nombre entier appelé 'pivot'. Les éléments de la liste sont placés dans une des deux sous-listes selon qu'ils sont inférieurs ou supérieurs au pivot. On remarque une 'astuce' importante : la fonction '`dividelist`' retourne deux listes, or une fonction ne peut retourner qu'un seul résultat. La solution est alors d'utiliser les tuples : la fonction '`dividelist`' retourne un tuple à deux éléments qui sont les deux sous-listes. Ainsi, en SCOL, les tuples servent principalement à deux choses : gérer les listes et grouper des éléments pour n'en faire plus qu'un, plus facile à manipuler.

La fonction '`dividelist`' n'est pas polymorphe, puisqu'elle suppose (par l'utilisation de la fonction `>`) qu'elle traite une liste d'entiers.

La fonction '`quicksort`' fonctionne alors sur un principe récursif très simple :

- la liste vide triée est toujours la liste vide
- si la liste n'est pas vide, alors, en prenant le premier élément de la liste pour pivot, la liste triée est la concaténation de :
  - la liste triée des éléments inférieurs au pivot
  - le pivot
  - la liste triée des éléments supérieurs au pivot

La fonction '`quicksort`' n'est pas polymorphe puisqu'elle appelle la fonction '`dividelist`'.

La fonction 'display' permet simplement de constituer une chaîne de caractère représentant la liste, où les éléments sont séparés par des '::' et qui se termine par 'nil'. Cette fonction n'est pas polymorphe, puisqu'elle suppose (par l'utilisation de la fonction 'itoa') qu'elle traite une liste d'entiers.

Le résultat de ce programme est : `2::3::5::6::8::nil`

Nous verrons plus loin une variante de 'quicksort' qui fonctionne sur des listes quelconques.

```
/* MyTest */

/* concatenation */
fun conc(p,q)=
 if p==nil then q
 else
  let p -> [a n]
  in a::conc n q;;

/* quicksort */

fun dividelist (x,p)=
 if p==nil then [nil nil]
 else
  let p->[a n] in
  let dividelist x n ->[r1 r2] in
  if x>a then [a::r1 r2]
  else [r1 a::r2];;

fun quicksort (l)=
 if l==nil then nil
 else
  let l->[vl nl]
  in let dividelist vl nl -> [va na]
     in conc quicksort va vl::quicksort na;;

/* display list */
fun display(l)=
 if l==nil then "nil"
 else
  let l->[a n]
  in strcat strcat (itoa a) "::" display n;;

fun mymain()=
 display quicksort 3::5::2::8::6::nil;;
```

3.5.4. Utilisation des tableaux

Les tableaux sont rarement utilisés dans un langage fonctionnel, on leur préfère souvent les listes, pour trois raisons :

- les tableaux se prêtent moins aux algorithmes récursifs
- les tableaux ne sont pas extensibles, on définit leur taille une fois pour toutes
- on ne peut empêcher le développeur d'écrire un programme qui essaie d'utiliser une case du tableau en dehors de ses limites. Cette erreur de dépassement d'indice n'est pas détectable à la compilation, elle est source d'erreur.

Cependant, le tableau offre un avantage sur la liste : l'accès en temps constant à n'importe quel élément.

La création d'un tableau se fait en particulier par la commande **mktab** qui prend en arguments la taille du tableau et une valeur d'initialisation.

L'accès au i-ème élément du tableau T se fait en écrivant : `T.i`

Si le tableau T est un tableau de tableaux, l'accès au j-ème élément du i-ème élément de T se fait en écrivant : `T.i.j` (on peut accumuler sans limite les indexations).

Pour modifier une valeur du tableau, on écrit simplement : `set T.i = ...`

3.5.5. Précisions sur certaines constructions de base

La fonction " `set X = V` " retourne la valeur `V` (le stockage de `V` dans `X` n'est qu'un effet de bord) .

La fonction " **`while`** `Condition` **`do`** `Expression` " calcule la condition (qui doit être un entier). Si cet entier est vrai, l'expression est calculée, et la condition est de nouveau évaluée, jusqu'à ce que celle-ci soit fausse. Elle retourne le résultat de la dernière expression calculée (`nil` si aucune expression n'a été calculée, c'est-à-dire si la condition était fausse dès la première évaluation).

La fonction " **`let`** `X -> N` **`in`** `Y` " calcule `X`, crée les variables locales contenues dans `N` puis calcule `Y` et retourne le résultat de `Y`. La portée statique des variables contenues dans `N` se limite à l'expression `Y`.

3.5.6. Utilisation des structures

Les structures s'utilisent un peu comme en C. Une structure est un type particulier contenant un ou plusieurs champs. Chaque champ est défini par un *nom de champ* et un type associé. Par exemple, une structure `Rec` contenant trois entiers et une chaîne s'écrit :

```
struct Rec = [xRec:I,yRec:I,zRec:I,nameRec:S] mkRec;;
```

Dans cet exemple, les noms `Rec`, `xRec`, `yRec`, `zRec`, `nameRec` et `mkRec` sont choisis par le développeur. La seule contrainte est que le nom du nouveau type (ici `Rec`) doit commencer par une majuscule.

Soit `x` un objet de type `Rec`, on accède aux différents champs en écrivant `x.xRec`, `x.yRec`, `x.zRec` ou `x.nameRec`. Les noms de champs sont considérés comme des fonctions : par exemple '`xRec`' est une fonction de type " `fun [ Rec ] I` ". Pour cette raison, si deux structures utilisent un même nom de champ, il y aura recouvrement : la seconde définition masquera la première.

Pour construire un objet de type `Rec`, on a besoin d'un constructeur : c'est le rôle de '`mkRec`' qui est ici une fonction de type " `fun [[I I I S]] Rec` ". Exemple :

```
fun main()=
 let mkRec [1 2 3 "abc"] -> r
      in r.xRec;;
```

Cette fonction retourne 1.

Pour modifier la valeur d'un champ, il suffit d'écrire :

```
      set r.xRec = ...
```

3.5.7. Utilisation des constructeurs de type

Il est intéressant dans certains cas de pouvoir utiliser une variable avec plusieurs types différents : dans la variable `x`, on veut avoir tantôt un entier, tantôt une chaîne de caractères, tantôt un tuple. Le

typage interdit ce genre d'opération. Pour permettre une telle manipulation, il faut utiliser l'équivalent de l'*union* du C : ce sont les constructeurs de type. Considérons un exemple :

```
typedef U =
   xU I
 | sU S
 | tU [I I]
 | nU ;;
```

Ceci définit un nouveau type U, qui peut contenir soit un entier, soit une chaîne, soit un tuple de deux entiers, soit rien. Les noms xU, sU, tU et nU sont appelés **constructeurs de type**. Ils sont considérés comme des fonctions. Par exemple, `xU` est une fonction de type " `fun [ I ] U` ".

On les appelle constructeurs, car eux-seuls permettent de construire un objet de type U. L'objet ainsi construit contient deux informations : le nom du constructeur qui a été utilisé, et la valeur utile. Le dernier constructeur nU est particulier car il n'utilise pas de valeur, on l'appelle **constructeur0**.

On traite un objet de type U grâce à la fonction `match`.

```
fun numconstr(x)=
match x with
 (xU u -> 0)
|(sU v -> 1)
|(tU [u v] ->2)
|(nU -> 3);;
```

Cette fonction prend un élément `x` de type U, et retourne respectivement `0, 1, 2` ou `3` pour `x` construit avec `xU`, `sU`, `tU` ou `nU`. Il est possible de définir une ligne des cas par défaut :

```
fun from_sU(x)=
  match x with
 (sU u -> 1)
|(_ -> 0);;
```

Cette fonction prend un élément `x` de type U et retourne 1 si `x` a été construit avec `sU`, 0 dans tous les autres cas.

Si le programme ne définit pas de cas par défaut (`_->…`), le compilateur " ajoute " la ligne `(_->nil)`.

La fonction `match` ne se contente pas de retrouver le constructeur d'une variable, elle permet aussi de récupérer la valeur de construction. L'exemple suivant le démontre :

```
/* MyTest */

typedef Node =
   Int I
 | Add [Node Node]
 | Mul [Node Node];;

fun EvalNode(n)=
 match n with
   (Int x -> x)
  |(Add [a b] -> (EvalNode a)+(EvalNode b))
  |(Mul [a b] -> (EvalNode a)*(EvalNode b));;

fun mymain()=
 itoa EvalNode Mul [Add [Int 1 Int 2] Int 3];;
```

Dans cet exemple, on définit un type Node qui permet de coder des arbres d'expressions contenant des constantes entières, des additions et des multiplications. La fonction `'EvalNode'` calcule la valeur d'un tel arbre.

Ici, la fonction `'mymain'` calcule l'arbre correspondant à l'expression : `(1+2)*3`

3.5.8. Manipulation des fonctions

SCOL permet de manipuler des fonctions au même titre que les entiers ou les chaînes de caractères. On utilise pour cela deux commandes du langage.

L'opérateur '@' permet de convertir un nom de fonction en objet fonction. Le compilateur considère en effet qu'un nom de fonction, non précédé de '@' représente un appel de la fonction, avec les paramètres qui suivent. Avec le symbole '@', le compilateur considère qu'un objet fonction doit être créé, en vue d'une manipulation ultérieure.

Pour que la manipulation de fonctions soit intéressante, il faut pouvoir appliquer un objet fonction à un ensemble d'arguments et calculer le résultat. On utilise pour cela la fonction **exec...with...**.Exemple :

```
fun baradd(x,y)= x+y;;
fun barmul(x,y)= x*y;;

fun foo(x,y,f)= exec f with [x y];;

fun main1()= foo 1 2 @baradd;;

fun main2()= exec @baradd with [1 2];;

fun main3()= baradd 1 2;;
```

Les trois fonctions `main1`, `main2` et `main3` retournent le même résultat.

Le type de la fonction `foo` est remarquable :

```
foo : fun [u0 u1 fun [u0 u1] u2] u2
```

En effet, la fonction `foo` prend deux arguments `x` et `y` quelconques et une fonction `f` qui n'est pas quelconque : c'est une fonction qui prend deux arguments du même type que `x` et `y` (types `u0` et `u1`). La fonction `foo` retourne un résultat de même type que la fonction `f`.

On remarque ici que la manipulation de fonctions en SCOL se fait de manière parfaitement contrôlée par le typage.

Une autre méthode de manipulation de fonctions consiste à créer une fonction à partir d'une autre fonction et d'un argument. Nous appellerons cela un nœud. SCOL offre une fonction qui réalise cette opération :

```
mknode : fun [fun [u0 u1] u2 u1]  fun [u0] u2
```

exemple :

```
fun f(x,y)=(atoi x)+y;;
fun g()= let mknode @f 1 -> h in exec h with ["16"];;
```

La fonction `g` définit une fonction `h` qui est égale à la fonction `f` dont on aurait fixé un argument (en l'occurrence le deuxième).

- le type de f est : `fun [ S I ] I`
- le type de h est : `fun [ S ] I`

Pour généraliser la fonction `mknode`, on définit les fonctions `mkfun1`, `mkfun2`, ..., `mkfun8`. Par exemple `mkfun8` prend une fonction à 8 arguments et un argument, et retourne une fonction à 7 arguments.

```
mkfun8 : fun [ fun [u0 u1 u2 u3 u4 u5 u6 u7] u8 u7] fun [u0 u1 u2 u3 u4 u5
u6] u8
```

En particulier la fonction **`mkfun2`** est équivalente à la fonction **`mknode`**.

3.5.9. Redéfinition de fonctions

Dans un fichier SCOL, on ne peut déclarer un nom qu'une seule fois. De plus, une fonction `f` référencée dans un fonction `g` doit être définie *en amont* de la fonction `g`.

Dans le cas où l'on compile successivement plusieurs fichiers SCOL, il est possible de redéfinir un nom défini dans un fichier précédent, à l'exception des noms de type.

Il peut être utile dans certains cas de prédéfinir le type d'une variable ou d'une fonction. C'est en particulier indispensable pour les variables mêlant des types autres que tuples, listes, entiers et chaînes de caractères. C'est également indispensable lorsque deux fonctions s'appellent mutuellement.

Exemple 1 :
```
/* definition d'une variable de type liste d'entiers */
typeof x = [ I r1 ];;
var x= [1 [2 [3 nil]]];;
/* sans le typeof, le langage déterminerait le type de x comme
étant [I [I [I u0]]], qui contient une variable libre */
```

Exemple 2 :
```
/* fonctions se référençant mutuellement */
proto g= fun[I] I;;

fun f(x)= if x>0 then 1+ g x-1 else 0;;
fun g(x)= if x>0 then 1+ f x-1 else 0;;
```

Si l'on souhaite initialiser une variable à **`nil`**, il est inutile d'écrire : `var x = nil;;`

`typeof` permet à la fois de prédéfinir un type et d'initialiser la variable à **`nil`**.

Exemple 3 :
```
/* définition d'une variable de liste d'entiers vide */
typeof x=[I r1];;
fun f()= set x = [1 [2 [3 nil]]];;
```

Dans un autre ordre d'idée, il est possible de définir une variable, un prototype, un constructeur de type ou une structure faisant appel à un type qui n'est pas encore défini. Il sera possible de définir ce type plus tard (dans un package ultérieur par exemple), mais une seule fois.

3.5.10. Nouveaux exemples

A présent vous connaissez tous les principes de calcul du langage SCOL. Nous allons présenter une variante du programme *Quicksort*, qui cette fois sera polymorphe, et qui permettra de supprimer les doublons (les éléments présents deux fois dans la liste).

Le principe est assez simple : on donne à la fonction `quicksort` une liste à trier ainsi qu'une fonction de comparaison de deux éléments de la liste.

Cette fonction qui a deux arguments (les deux éléments à comparer) doit retourner :

- un nombre strictement positif si le premier élément est supérieur au deuxième
- un nombre strictement négatif si le premier élément est inférieur au deuxième
- zéro si les deux éléments sont égaux et que l'on souhaite supprimer les doublons de la liste. Si l'on ne souhaite pas supprimer les doublons, ils suffit de retourner n'importe quel entier non nul lorsque deux éléments sont égaux.

Le programme suivant effectue deux tris, un sur une liste d'entiers, un autre sur une liste de chaînes de caractères. Les fonctions d'affichage ne sont pas polymorphes : il y en a une pour afficher les listes d'entiers, une autre pour afficher les listes de chaînes de caractères.

Le type de la fonction `quicksort` vaut ici :

```
fun [[u0 r1] fun [u0 u0] I] [u0 r1]
```

On notera dans la fonction de comparaison `mystrcmp` l'utilisation de la fonction `strcmp`. Cette fonction, que l'on retrouve dans la plupart des langages de programmation, permet de comparer deux chaînes de caractères. Elle retourne 1, si la première est supérieure (dans l'ordre alphabétique) à la seconde, -1 si elle est inférieure, 0 si les deux chaînes sont égales.

```
/* MyTest */

fun conc(p,q)=
 if p==nil then q
 else (hd p)::conc (tl p) q;;

fun dividelist (x,p,f)=
 if p==nil then [nil nil]
 else
  let p->[a n] in
  let dividelist x n f ->[r1 r2] in
  let exec f with [a x] -> r in
  if r==0 then [r1 r2]
  else if r<0 then [a::r1 r2]
  else [r1 a::r2];;

fun quicksort(l,f)=
 if l==nil then nil
 else let l->[vl nl] in
 let dividelist vl nl f->[va na] in
 conc quicksort va f vl::quicksort na f;;

/* display list */
fun displayIntList(l)=
 if l==nil then "nil\n"
 else
  let l->[a n]
  in strcat strcat (itoa a) "::" displayIntList n;;

fun displayStrList(l)=
 if l==nil then "nil\n"
 else
  let l->[a n]
  in strcat strcat a "::" displayStrList n;;

fun myintcmp(a,b)=a-b;;
fun mystrcmp(a,b)=strcmp a b;;

fun mymain()=
```

```
strcat
 displayIntList
  quicksort 3::5::2::8::6::5::nil @myintcmp
 displayStrList
  quicksort "ab"::"abc"::"xy"::"www"::"pqr"::nil @mystrcmp;;
```

Le programme retourne le résultat suivant :

```
2::3::5::6::8::nil
ab::abc::pqr::www::xy::nil
```

On remarquera effectivement que les doublons ont été supprimés : le nombre 5 apparaissait deux fois dans la liste à trier.

3.5.11. Librairie standard

On trouvera ici la liste des fonctions de base du langage SCOL. Pour chaque fonction, on indiquera le type. Elles sont réparties en plusieurs catégories :

- fonctions sur les entiers
- fonctions sur les chaînes de caractères
- fonctions sur les listes
- fonctions sur les tableaux
- fonctions sur les nombres flottants
- fonctions de temps
- fonctions de console

Outre les fonctions classiques de tout langage de programmation, on remarquera particulièrement les fonctions suivantes :

- `zip` et `unzip` : compression et décompression d'une chaîne de caractères
- `strextr` et `strbuild` : découpage d'une chaîne en liste de lignes, chacune étant une liste de mots. Ceci simplifie tous les problèmes de parsing
- `_getlongname` : fonction de hachage (ou de signature) d'une chaîne de caractères

3.5.11.1. Fonctions sur les entiers

`rand : fun [] I`
retourne un entier aléatoire, compris entre 0 et 32767

`srand : fun [I] I`
initialise la série aléatoire avec un entier. retourne 0.

`max : fun [ I I ] I`
retourne le max de deux entiers

`min : fun [ I I ] I`
retourne le min de deux entiers

`abs : fun [ I ] I`
retourne la valeur absolue d'un entier

`mod : fun [ I I ] I`
retourne le reste de la division d'un entier par un autre

3.5.11.2. Fonctions sur les chaînes de caractères

`strlen : fun [ S ] I`

retourne la taille d'une chaîne de caractères.

Par exemple, `strlen "12abc345de"` retourne l'entier 10

`strcat : fun [ S S ] S`
concatène deux chaînes de caractères (les deux chaînes initiales ne sont pas modifiées)

Par exemple, `strcat "12" "abc"` retourne la chaîne "`12abc`"

`strcatn : fun [[S r1]] S`
concatène une liste de chaînes de caractères (équivalente, mais en plus efficace, à la fonction précédente répétée *n-1* fois)

Par exemple, `strcatn "12"::"abc"::"345"::"de"::nil` retourne la chaîne "`12abc345de`"

`strcmp : fun [ S S ] I`
compare deux chaînes de caractères (en utilisant la fonction C standard). Retourne 0 si les deux chaînes sont identiques, 1 ou –1 sinon.

`strfind : fun [ S S I ] I`
recherche la première chaîne dans la seconde à partir de la position passée en paramètre (le premier caractère est en position 0). Retourne `nil` si la chaîne n'a pas été trouvée, ou bien la position à laquelle la chaîne a été trouvée.

`strfindi : fun [ S S I ] I`
idem précédent, mais sans tenir compte des différences entre majuscules et minuscules.

`listtostr : fun [ [I r1] ] S`
transforme une liste d'entiers (considérés comme des codes Ascii) en chaîne de caractères : chaque maillon donne un caractère. Le premier maillon donne le premier caractère.

Par exemple `listtostr 65::66:67::nil` retourne la chaîne "`ABC`"

`strtolist : fun [ S ] [ I r1 ]`
fonction inverse de la précédente.

Par exemple `strtolist "ABC"` retourne la chaîne `65::66::67::nil`

`atoi : fun [ S ] I`
interprète une chaîne de caractères comme un entier.

Par exemple, `atoi "53"` retourne l'entier `53`

`itoa : fun [ I ] S`
fonction inverse de la précédente.

Par exemple, `itoa 53` retourne la chaîne "`53`"

`ctoa : fun [ I ] S`
crée une chaîne de caractères contenant un seul caractère de code ascii donné.

Par exemple, `ctoa 65` retourne la chaîne "`A`"

`htoi : fun [ S ] I`
interprète une chaîne de caractères comme un entier codé en hexadécimal (non signé)

`itoh : fun [ I ] S`
fonction inverse de la précédente.

substr : fun [S I I] S
retourne une sous-chaîne de la chaîne passée en argument.

Le premier entier donne la position de départ de la sous-chaîne, le premier caractère étant en position 0, et le second entier en donne la taille.

Par exemple, `substr "abcdef" 2 3` retourne la chaîne "`cde`"

strdup : fun [S] S
crée une copie en mémoire d'une chaîne de caractères. Ceci n'est intéressant que lorsqu'on modifie "sur place" une chaîne de caractères au moyen de la fonction `set_nth_char` définie plus bas.

strlowercase : fun [S] S
crée une copie d'une chaîne en remplaçant les majuscules par des minuscules

struppercase : fun [S] S
crée une copie d'une chaîne en remplaçant les minuscules par des majuscules

strcmpi : fun [S S] I
compare deux chaînes de caractères sans tenir compte des majuscules-minuscules (en utilisant la fonction C standard)

nth_char : fun [S I] I
retourne le n-ème caractère d'une chaîne

set_nth_char : fun [S I I] S
modifie le n-ème caractère d'une chaîne (danger : la chaîne est modifiée 'sur place', tous les pointeurs vers la chaîne sont affectés par cette modification, il s'agit d'un effet de bord)

zip : fun [S] S
effectue la compression d'une chaîne de caractères. Le taux de compression est en moyenne de 60\%.

unzip : fun [S] S
fonction inverse de la fonction précédente.

strtoweb : fun [S] S
fonction convertissant une chaîne de caractères quelconque en une chaîne ne comportant que des symboles alphanumériques, ainsi que des '+' et des '%' :

- les caractères alphanumériques sont conservés
- les espaces sont remplacés par des '+'
- les autres caractères sont remplacés par un '%' suivi de 2 chiffres hexadécimaux

webtostr : fun [S] S
fonction inverse de la précédente

_getlongname : fun [S1 S2 S3] S
cette fonction est utilisée dans la gestion des fichiers, mais elle s'avère pratique dans d'autres utilisations. Elle réalise une signature.

- S1 est la chaîne à signer,
- S2 est une chaîne quelconque
- S3 code le type de signature, en reprenant le caractère spécifique
- "#" : premier type de signature

Cette fonction retourne la concaténation de S2, du caractère spécifique de la signature, et de la signature de la chaîne S1.

`lineextr : fun [S] [S r1]`
La fonction `lineextr` découpe le texte initial en lignes (les lignes sont séparées par les caractères 10 ou 13).

`linebuild : fun [[S r1]] S`
La fonction `linebuild` est la fonction inverse de la précédente. Elle reconstitue le texte en concaténant les lignes, séparées d'un caractère 10.

`strextr : fun [S] [[S r1] r1]`
La fonction `strextr` découpe le texte initial en lignes, puis chaque ligne en mots, et construit le résultat sous forme d'une liste de lignes, chaque ligne étant une liste de mots. Seules les lignes contenant au moins un mot sont retenues.

Il est possible de mettre des caractères spéciaux dans les mots, à l'aide du caractère `\` :

- `\\` pour le caractère \
- `\+' '` (backslash suivi d'un espace) pour le caractère espace
- \+nombre décimal d'au plus 3 chiffres pour n'importe quel code ascii

Un `\` en fin de ligne permet d'ignorer le retour à la ligne.

Par exemple, `strextr "abc def\n1 23 456"` retourne la double liste ("abc"::"def"::`nil`)::("23"::"456"::`nil`)::`nil`

`strbuild : fun [[[S r1] r1]] S`
La fonction `strbuild` est la fonction inverse de la fonction précédente. Elle reconstitue le texte de base, en remplaçant les caractères spéciaux par leur équivalent avec \

3.5.11.3. Fonctions sur les listes

`constructeur de liste ' ::'`
opérateur de construction de liste. Par exemple pour construire la liste des 5 premiers entiers :

```
1 :: 2 :: 3 :: 4 :: 5 :: nil
```

`hd : fun [[u0 r1]] u0`
retourne le premier élément d'une liste.

`tl : fun [[u0 r1]] [u0 r1]`
retourne la liste sans le premier élément.

`sizelist : fun [[u0 r1]] I`
retourne la taille de la liste.

`nth_list : fun [[u0 r1] I] u0`
retourne le n-ième élément d'une liste (les éléments sont numéroté à partir de 0). Si l'élément n'existe pas, `nil` est retourné.

Par exemple `nth_list 3::5::4::6::nil 2` retourne l'entier 4.

`endlist : fun [[u0 r1] I] [u0 r1]`
retourne la liste à partir du n-ième élément (les éléments sont numérotés à partir de 0). Si l'élément n'existe pas, `nil` est retourné.

Par exemple `endlist 3::5::4::6::nil 2` retourne la liste 4::6::`nil`.

3.5.11.4. Fonctions sur les tableaux

`sizetab : fun [tab u0] I`
retourne la taille d'un tableau quelconque (nombre de cases du tableau)

`mktab : fun [ I u0 ] tab u0`
crée un tableau de taille quelconque initialisé avec une valeur quelconque.

`tabtolist : fun [ tab u0 ] [u0 r1]`
crée une liste à partir d'un tableau, en prenant les éléments dans l'ordre d'indice croissant.

`tabtolistR : fun [ tab u0 ] [u0 r1]`
crée une liste à partir d'un tableau, en prenant les éléments dans l'ordre d'indice décroissant.

`listtotab : fun [ [u0 r1] ] tab u0`
crée un tableau à partir d'une liste, en conservant l'ordre des éléments de la liste

`listtotabR : fun [ [u0 r1] ] tab u0`
crée un tableau à partir d'une liste, en inversant l'ordre des éléments de la liste

3.5.11.5. Fonctions sur les nombres flottants

`itof : fun [ I ] F`
transforme un entier en flottant

`ftoi : fun [ F ] I`
transforme un flottant en entier (en arrondissant le flottant)

`ftoa : fun [ F ] S`
transforme un flottant en chaîne de caractères

`atof : fun [ S ] F`
fonction inverse de la précédente.

`absf : fun [ F ] F`
calcule la valeur absolue d'un flottant

`PIf : fun [ ] F`
retourne la constante **pi**

`cos : fun [ F ] F`
fonction cosinus

`sin : fun [ F ] F`
fonction sinus

`tan : fun [ F ] F`
fonction tangente

`acos : fun [ F ] F`
fonction arccosinus

`asin : fun [ F ] F`
fonction arcsinus

atan : fun [F] F
fonction arctangente

atan2 : fun [F F] F
fonction arctangente à deux arguments : " `atan2 y x` " retourne l'angle signé formé entre l'axe des abscisses et le point (x,y).

Ef : fun [ ] F
retourne la constante **e**.

log : fun [F] F
fonction logarithme

log10 : fun [F] F
fonction logarithme en base 10

exp : fun [F] F
fonction exponentielle

pow : fun [F F] F
fonction puissance : ``pow x y'' retourne x élevé à la puissance y.

sqr : fun [F] F
calcul du carré d'un flottant

sqrt : fun [F] F
calcul de la racine carrée d'un flottant

rootn : fun [F F] F
calcul de la racine n-ième d'un flottant : " `rootn x y` " retourne x à la puissance 1/y.

3.5.11.6. Fonctions d'heure

time : fun [] I
retourne le nombre de secondes écoulées depuis le 1er Janvier 1970.

ctime : fun [I] S
retourne une chaîne de caractères donnant l'heure et la date au format suivant : ``Tue Jan 21 11:24:53 1997" . Le paramètre est le nombre de secondes écoulées depuis le 1er Janvier 1970, typiquement issu de la fonction précédente

_tickcount : fun [] I
retourne le nombre de millisecondes écoulées depuis le lancement de la machine

3.5.11.7. Fonctions de console

_showconsole : fun [ ] I

affiche la console

_hideconsole : fun [ ] I

cache la console

_fooS : fun [S] S

envoi d'une chaîne de caractères vers la console. retourne la même chaîne. La chaîne ne doit pas faire plus de 4Ko.

`_fooI : fun [I] I`

envoi d'un entier en hexadécimal vers la console. retourne le même entier.

`_fooIList : fun [[I r1]] [I r1]`

envoi d'une liste d'entiers en hexadécimal vers la console. résultat inchangé.

`_fooSList : fun [[S r1]] [S r1]`

envoi d'une liste de chaînes vers la console. résultat inchangé.

3.6. Variables globales de la machine SCOL

La machine SCOL gère une liste de variables de ressources, que l'on appellera simplement '**ressources**'. Ces variables sont définies par leur nom, et sont associées à une chaîne de caractères. Elle sont indépendantes de la gestion des canaux, et notamment survivent au canal qui les a définies.

L'API contient deux instructions :
`_getress : fun [ S ] S`
cherche une ressource d'un certain nom et retourne la valeur associée (**`nil`** si la ressource n'est pas définie)

`_setress : fun [ S1 S2 ] S`
définit la ressource S_1 avec la valeur S_2 et retourne S_2. Si la ressource n'existait pas, elle est créée. Si elle existait, une nouvelle valeur lui est affectée. Si S_2 vaut `nil`, la ressource est détruite.

SCOL utilise un fichier d'initialisation des ressources, il s'appelle **usmress.ini**, et se trouve dans le répertoire SCOL (typiquement *C:/Program Files/SCOL*). C'est un fichier texte contenant des lignes de deux mots, le premier étant le nom d'une ressource, le second étant la valeur correspondante.

Ce fichier est analysé au début du fonctionnement de la machine SCOL grâce à la fonction suivante :

```
fun multiress(res)=
 if res==nil then 0
 else let res ->[[l n] nxt] in
 (if strcmp l "#" then _setress l hd n else nil;
  multiress nxt);;

 ...
 multiress strextr _loadressini;
 ...
```

On remarque la fonction `_loadressini` :
`_loadressini : fun [ ] S`
lit le fichier **usmress.ini** et le retourne dans une chaîne de caractères. En effet, ce fichier n'est a priori pas dans une des partitions SCOL, c'est pourquoi il est nécessaire d'avoir une fonction spéciale pour le lire.

Il y a deux variables particulières que l'on peut considérer comme des ressources, il s'agit du numéro et du nom de version de la machine SCOL :
`_version : fun [ ] I`
retourne le numéro de version

`_versionname : fun [ ] S`
retourne le nom de la version

L'ordre des chapitres suivants n'est pas déterminant. A vous de le déterminer en fonction de vos centres d'intérêt : si vous vous intéressez aux particularités du langage SCOL commencez plutôt par lire les parties sur les canaux et communications et sur les fichiers.

Si vous êtes pressé de développer vos premiers programmes, lisez d'abord la partie sur la programmation événementielle.

4. CANAUX ET COMMUNICATIONS

En SCOL, la notion de canal est fortement liée à celle de communication. En effet, les liaisons réseaux sont systématiquement liées à un environnement pour former un canal. Dans la première partie nous expliquons comment créer et manipuler les canaux. Dans la seconde partie, nous montrons comment utiliser ces canaux pour communiquer d'une machine à l'autre.

4.1. Manipulation des canaux

On a vu au début du chapitre précédent que la machine SCOL reposait sur la notion de canal. La machine SCOL est multi-canaux. Chaque canal est un couple (environnement, liaison réseau). La liaison réseau est typiquement une socket Tcp-Ip ou Udp. Mais elle peut être inexistante : le canal est alors dit **unplugged**.

Canaux et environnements sont des objets manipulables par SCOL. Deux types leur sont donc associés :

- le type **`Chn`** représente un canal.
- le type **`Env`** représente un environnement.

4.1.1. API de manipulation des canaux

`_channel : fun [] Chn`
retourne le canal courant.

`_script : fun [S] I`
exécution d'un script dans l'environnement du canal courant. Chaque ligne du script est de la forme commande-arguments : la portée est définie dynamiquement. Le résultat vaut 0. Une commande inconnue, ou un mauvais typage des paramètres ne sont pas considérés comme des erreurs : la commande est simplement sautée. La syntaxe exacte d'un script est définie plus loin, mais on peut déjà préciser que c'est la même que celle des fichiers **.SCOL* (voir les exemples '*Hello World*' du chapitre II)

`_scriptc : fun [Chn S] I`
idem précédent, mais dans un canal différent qui est précisé en paramètre.
Remarque : `_script X` est équivalent à `_scriptc _channel X`

`_load : fun [S] I`
chargement et compilation dans le canal courant, d'un fichier SCOL (appelé aussi package), dont le nom est passé en paramètre (voir plus loin la gestion des fichiers).

`_setchannel : fun [Chn] Chn`
change le canal courant (à utiliser avec précaution).

4.1.2. API de gestion des environnements

`_envchannel : fun [ Chn ] Env`
retourne l'environnement associé à un canal.

`_removepkg : fun [ Env ] Env`

retourne l'environnement dont on a retiré le package situé en tête (c'est-à-dire le dernier package compilé)

`_envfirstname : fun [ Env ] S`
retourne le nom du premier package de cet environnement. En l'utilisant plusieurs fois avec `_removepkg`, cette fonction permet de connaître le nom de tous les packages d'un environnement.

`_setenv : fun [ Chn Env ] I`
change l'environnement d'un canal. Si l'environnement vaut `nil`, le canal reprendra l'**environnement minimal**.

4.1.3. Création et destruction d'un canal

4.1.3.1. Canal Tcp-Ip

Un canal étant un couple (environnement, liaison réseau), pour créer un canal, il faudra préciser les deux éléments du couple. La fonction à utiliser est `_openchannel`. Cette fonction permet de créer un canal unplugged ou bien un canal avec une liaison Tcp-Ip.

Cette fonction utilise trois arguments.

- l'adresse de connexion de la liaison (`nil` si le canal est **unplugged**). L'adresse est une chaîne comprenant l'**adresse IP** du serveur à contacter et le numéro de port, par exemple : " 123.234.54.34:1025 ". Si l'adresse IP est omise, la connexion se fait localement (" :1025 " est équivalent à " 127.0.0.1:1025 ")
- un script (la syntaxe est définie plus loin). `Nil` est possible, et est équivalent ici à la chaîne vide.
- l'environnement dont le canal va hériter. Si l'environnement vaut `nil`, le canal hérite de l'**environnement minimal**.

La fonction retourne le canal créé. Celui-ci vaut `nil` si une erreur s'est produite.

`_openchannel : fun [ S S Env ] Chn`

Le fonctionnement de `_openchannel` est le suivant : un nouveau canal est créé avec l'environnement passé en paramètre et l'éventuelle liaison réseau. Puis le script est exécuté sur ce nouveau canal.

Exemple : vous voulez créer un canal unplugged qui hérite de l'environnement du canal courant, et dans lequel vous voulez compiler le package "*new.pkg*" et exécuter la fonction `main` qui se trouve dans ce package. Il suffit d'écrire :

```
_openchannel nil "_load \"new.pkg\"\nmain" _envchannel _channel
```

La version suivante est équivalente :

```
_scriptc
 (_openchannel nil "" _envchannel _channel)
 "_load \"new.pkg\"\nmain"
```

Pour ouvrir un canal vers le port 2000 de la machine dont l'adresse Ip est 1.2.3.4, lui donner l'environnement minimal, y compiler le package "*new.pkg*" et exécuter la fonction `main` qui se trouve dans ce package, il suffit d'écrire :

```
_openchannel "1.2.3.4:2000" "_load \"new.pkg\"\nmain" nil
```

4.1.3.2. Canal Udp

Il est possible de créer un canal dont la liaison réseau est une socket UDP en écoute :
`_setUDP : fun [ Env I S ] Chn`

Cette fonction prend un environnement, un numéro de port et un script. Un canal UDP est créé sur le port dont on donne le numéro, avec l'environnement donné dans lequel on commence par exécuter le script.

Rappelons que les messages UDP n'offrent aucune garantie : la transmission et l'ordre d'arrivée ne sont pas assurés. Par contre, il n'y a pas de bufferisation : la latence de transmission est donc faible.

Le canal UDP n'est pas connecté à un autre canal : plusieurs correspondants peuvent émettre des messages vers ce canal, et il y a même possibilité de broadcast.

4.1.3.3. Destruction de canaux

`_closechannel : fun [] I`
ferme le canal courant.

`_killchannel : fun [Chn] I`
ferme le canal spécifié.

`_closemachine : fun [] I`
ferme tous les canaux, et donc arrête la machine SCOL.

4.1.4. Création et destruction d'un serveur

Nous venons de voir comment ouvrir des canaux Tcp-Ip, grâce à la fonction `_openchannel`. Pour cela, nous avons vu qu'il fallait préciser l'adresse Ip et le numéro de port du correspondant. Cela signifie que le correspondant "attend" ce type de connexion. En terminologie réseau, on dit que le correspondant a ouvert un **serveur** sur ce port. Evidemment, le langage SCOL permet d'ouvrir et de manipuler de tels serveurs. Ce sont des objets de type `Srv`.

Pour définir un serveur, on utilise la fonction suivante :

`_setserver : fun [Env I S] Srv`

ouvre un serveur sur le numéro de port et avec le script passés en paramètres. retourne `nil` si impossible (le port est probablement déjà utilisé par un autre serveur). Le serveur ainsi créé hérite de l'environnement passé en paramètre.

Lorsqu'une connexion est demandée vers ce serveur, c'est-à-dire lorsqu'une autre machine a exécuté la fonction `_openchannel` avec l'adresse de ce serveur, un nouveau canal est créé. Ce canal hérite de l'environnement défini avec le serveur. Le script défini avec le serveur est alors immédiatement exécuté dans ce nouveau canal.

Une fois créé, un canal serveur est indifférenciable d'un canal client. C'est un des concepts clefs de SCOL : une fois la connexion effectuée, la communication est symétrique.

Supposons par exemple qu'on appelle sur la machine d'Alice la fonction suivante :

```
_setserver _envchannel _channel 2000 "_load \"new.pkg\"\nmain"
```

Un serveur est créé sur le port 2000. L'environnement défini avec le serveur est l'environnement courant. Le script défini avec le serveur compile le fichier "*new.pkg*" et exécute la fonction `main`.

Lorsqu'une connexion est ouverte depuis une autre machine, celle de Bob par exemple, un canal est créé sur la machine d'Alice, qui hérite de l'environnement et dans lequel on exécute le script. Alors, il y

a sur la machine d'Alice un canal qui est lié à un canal de la machine de Bob. Cette liaison est symétrique et permettra l'échange de données entre les deux machines.

Pour arrêter un serveur, on utilise la fonction suivante :

```
_closeserver : fun [Srv] I
```

détruit le serveur passé en paramètre.

Attention : ceci supprime seulement la possibilité de recevoir de nouveaux appels sur ce serveur, cela ne détruit pas les canaux qui ont été créés à partir de lui.

4.1.5. Fonctions supplémentaires de gestion des canaux

`_channelname : fun [Chn] S`
retourne l'adresse complète du correspondant d'un canal (adresse IP et numéro de port) : par exemple " 127.0.0.1:1234 "

`_channelIP : fun [Chn] S`
retourne uniquement l'adresse IP du correspondant : dans l'exemple précédent, uniquement " 127.0.0.1 "

`_channelport : fun [Chn] I`
retourne le numéro de port du correspondant : dans l'exemple précédent, 1234.

`_gethostbyname : fun [S] S`
fonction usuelle effectuant la résolution d'adresse dans le sens *nom de machine->adresse IP*. Attention, cette fonction est **bloquante** : elle peut arrêter la machine pendant plusieurs secondes si la résolution est difficile ou impossible.

`_getnamebyIP : fun [S] S`
fonction usuelle effectuant la résolution d'adresse dans le sens *adresse IP->nom de machine*. Attention, cette fonction est **bloquante** : elle peut arrêter la machine pendant plusieurs secondes si la résolution est difficile ou impossible.

`_hostname : fun [] S`
retourne le nom de l'hôte. si l'hôte n'a pas de nom défini, retourne ``localhost''.

`_hostIP : fun [] S`
retourne l'adresse IP de l'hôte : si l'hôte n'en a pas, retourne " 127.0.0.1 "

`_channeltime : fun [ Chn ] I`
retourne le temps écoulé depuis la création du canal (en secondes)

`_servertime : fun [ Srv ] I`
retourne le temps écoulé depuis la création du serveur (en secondes)

4.1.6. Syntaxe des scripts

Comme annoncé, nous précisons ici la syntaxe des scripts. Ceux-ci sont utilisés dans plusieurs cas :

- les fichiers **.SCOL* de démarrage de la machine SCOL
- les fonctions `_script` et `_scriptc`.
- les fonctions de création de canaux et de serveurs, comme scripts d'initialisation d'environnement.

Un script est une chaîne de caractères se terminant par le caractère 0.

Un script est composé d'une ou plusieurs lignes : les lignes sont séparées par des caractères 10.

Chaque ligne est constituée de mots séparés par des caractères de code inférieur ou égal à 32.

Le premier mot est un mnémonique qui correspond normalement à un nom de fonction.

Les mots suivants sont :

- soit `NIL`
- soit un entier en hexadécimal (au plus 8 chiffres)
- soit une chaîne de caractères commençant et finissant par un `"`

Dans la chaîne de caractères, le caractère backslash `(\)` a une signification particulière :

- `\n` : caractère 10
- `\z` : caractère 0
- `\nombre en décimal (au plus trois chiffres)` : caractère de code quelconque
- `\autre` : le backslash est ignoré et on saute au prochain caractère de code supérieur ou égal à 32

Ainsi pour le caractère `\`, on utilisera `\\`. Pour le caractère `"`, on utilisera `\"`.

Chaque ligne de script est interprétée comme une commande suivie d'arguments. Ces arguments sont soit des entiers, soit des chaînes de caractères, soit `NIL` (seule possibilité pour les autres types).

Attention :

les fichiers **.SCOL* ne doivent pas être trop longs : leur taille est en effet limitée par la taille de la ligne de commande acceptée par *Windows*. Limitez-vous à trois ou quatre lignes. Dans les autres utilisations des scripts, il n'y a pas de taille limite au script.

Exemple de fichier script **.SCOL* :

```
_load "foo.pkg"
main 1234 "qsd\\q\"sdf\n\0\12\234\0122" NIL
```

Même script utilisé avec la fonction `_script` :

```
_script "_load \"foo.pkg\"\nmain 1234
\"qsd\\\\q\\\"sdf\\n\\0\\12\\234\\0122\" NIL"
```

4.2. Communications en SCOL

4.2.1. Contrôle des connexions : événements particuliers

Ouvrir un canal avec la fonction `_openchannel` est une opération qui prend un certain temps (jusqu'à quelques secondes). L'ordinateur doit trouver son chemin à travers le réseau, puis communiquer avec l'autre machine pour vérifier qu'un serveur est effectivement ouvert. Une question se pose : quand la liaison est-elle réellement établie ?

En fait la fonction `_openchannel` est pratiquement instantanée : le canal est crée immédiatement, et la connexion réseau est lancée, puis le programme continue avant que la connexion ne soit véritablement effective.

Lorsque la connexion est réalisée, la machine SCOL recherche une fonction de nom '`_connected`' de type :

```
fun [ ] ?
```

(Le point d'interrogation signifie que la machine ne se soucie pas du type résultat de la fonction `_connected`)

Si la fonction `_connected` n'a pas été définie, rien ne se passe, la machine continue normalement, sinon la fonction `_connected` est exécutée.

La même chose se produit sur le serveur : un canal est créé sur la machine qui abrite le serveur, le script y est exécuté, puis l'éventuelle fonction `_connected` y est déclenchée.

Un autre événement réseau peut se produire de manière imprévisible : la déconnexion. Celle-ci est due soit à une destruction du canal par le correspondant, soit à un arrêt de la machine du correspondant, soit encore à un problème réseau.

Dans ce cas, la machine SCOL va fermer le canal qui a perdu son correspondant, mais juste avant, elle recherche une fonction de nom '`_closed`' et de type :

```
fun [ ] ?
```

Si la fonction `_closed` n'a pas été définie, rien ne se passe, la machine continue normalement, sinon la fonction `_closed` est exécutée.

De même, si le serveur arrive à saturation de nombre de connexions simultanées (10 avec la version gratuite de la machine virtuelle SCOL), la fonction `_fullserver` sera exécutée dans l'environnement du serveur, et le message '__FullServer' sera transmis au client, juste avant que la connexion ne soit fermée.

4.2.2. Envoi d'un message par la fonction `_on`

Une fois un canal créé, des messages peuvent transiter dans les deux sens. Les messages qui circulent sont toujours de la forme " commande arguments ". En fait, ils ont la forme d'un script à une seule ligne. Lorsqu'un tel message arrive, la machine SCOL recherche dans l'environnement du canal une fonction dont le nom est " `__commande` " (le *double* underscore est ajouté par la machine réceptrice et garantit donc que seules les fonctions commençant par un double underscore pourront être activées par le correspondant). Si une telle fonction existe, la machine SCOL vérifie le type des arguments. Si tout correspond, la fonction est exécutée et le résultat est perdu : seuls compteront les effets de bord.

Pour utiliser le canal dans le sens de l'émission, le langage SCOL offre une solution simple. On prendra l'exemple suivant : Bob veut envoyer sur le canal `AliceChannel` la commande `foo` avec trois paramètres : deux entiers 123 et 345 et une chaîne "`bar`".

La fonction à utiliser s'appelle "`_on` ", elle est de type : `fun[Chn Comm] I`.

Remarque : les messages envoyés sur un canal **unplugged** sont simplement ignorés.

Avec la fonction `_on`, l'exemple précédent s'écrit :

```
defcom X = foo I I S;;
...
  _on AliceChannel X [123  234 "bar"];
...
```

Lors du **`defcom`**, le compilateur définit un **constructeur de communication** X dont le type est :

```
fun[[I I S]] Comm
```

X est donc une fonction qui prend un tuple de trois éléments (un entier, un entier puis une chaîne) et retourne un message contenant ces trois arguments précédés de la commande " `foo` ". Les constructeurs de communications permettent uniquement de passer des entiers (I) et des chaînes de caractères (S).

Définissons la fonction `_on` :

```
_on :  fun[Chn Comm] I .
```

Pour simplifier l'écriture, on peut parfois donner le même nom à la commande et au constructeur de communication :

```
defcom foo = foo I I S;;
...
  _on AliceChannel foo [123  234 "bar"];
...
```

Attention : les messages produits par les constructeurs de communication en vue d'être transmis par la fonction `_on` ne devraient pas être trop longs (moins de 8Ko).

Il existe une variante avec **`defcomvar`**. Cette variante est notamment intéressante pour l'utilisation des " call-backs ". Soit par exemple une fonction " `call` " qui prend une chaîne et un entier en entrée et doit envoyer un message avec un argument qui est l'entier et une commande qui est la chaîne. La définition statique du nom de commande avec `defcom` ne permet pas de résoudre ce cas. La fonction defcomvar doit alors être utilisée :

```
defcomvar Y = I;;
fun call(s,i)=
  _on AliceChannel Y s [i];;
```

La fonction `Y` ici créée a le type " `fun[S [I]] Comm`". La chaîne qu'elle attend sera utilisée comme commande.

L'exemple du paragraphe précédent s'écrit aussi :

```
defcomvar Y = I I S;;
...
  _on AliceChannel Y "foo" [123  234 "bar"];
...
```

4.2.3. Contrôle des files d'attente des messages

Lorsqu'un message est émis par la fonction **`_on`** sur un canal de type Tcp-Ip, il est placé dans une file d'attente propre au canal. La machine SCOL tente de l'envoyer aussi vite que possible. Il est utile de pouvoir contrôler la taille de la file d'attente (appelée fifo : First In First Out) grâce à deux fonctions :

`_waitingfifo : fun [ Chn ] I`
retourne le nombre de messages présents dans la file d'attente.

`_sizewaitingfifo : fun [ Chn ] I`
retourne la taille en octets de tous les messages présents dans la file d'attente.

`_setsizefifo : fun [ Chn I ] Chn`
définit une taille maximale pour les fifos (`nil` : pas de taille limite). Si lors d'un _on, la taille de la fifo est dépassée, le canal est déconnecté, et un événement _closed sera produit.

Remarque : la fifo SCOL est une fifo qui se trouve entre l'application SCOL et la fifo de la couche réseau de l'ordinateur, dont la taille n'est pas facilement calculable.

La limitation de la taille des fifo permet principalement de protéger la machine SCOL d'un manque de mémoire, car les fifos des canaux et les applications SCOL cohabitent dans la même bande mémoire.

4.2.4. Emission d'un message Udp

Nous avons vu comment créer un canal Udp avec la fonction `_setUdp`. Ce canal est en fait un serveur Udp : il écoute sur un certain port les messages Udp qu'on lui transmettra. Ce messages sont aussi des objets de type `Comm`, et lorsqu'un tel message arrive, la machine SCOL recherche dans

l'environnement du canal une fonction dont le nom est " `__commande` ", exactement comme avec la fonction `_on`.

Pour envoyer un message vers un canal UDP, on utilise :

`_sendUDP : fun [ S Comm ] I`

Il est également possible d'envoyer un message UDP via un canal UDP (créé par **`_setUDP`**), en utilisant la fonction suivante. Ceci peut être utile pour franchir certains proxies dans le sens extérieur vers intérieur : en effet, certaines passerelles se souviennent qu'un certain " serveur " UDP interne a transmis un message vers un certain serveur UDP externe, et autorisent alors celui-ci à répondre, ce qui permet à des messages UDP de franchir la passerelle dans le sens extérieur vers intérieur.

`_sendUDPchn : fun [ Chn S Comm ] I`

La chaîne contient l'adresse du correspondant (au même format que pour `_openchannel`). On peut utiliser une adresse broadcast.

Contrairement à un canal Tcp-Ip, un canal Udp n'est pas associé à une machine unique : tout le monde peut lui envoyer un message. Il peut être utile au canal Udp de connaître l'adresse Ip de celui qui lui a envoyé un message. Pour cela, il suffit simplement d'utiliser la fonction `_channelIP` décrite plus haut. En effet, la valeur que cette fonction retourne est remise à jour à chaque réception d'un message.

4.2.5. Autre utilisation des constructeurs de communication

Nous avons mentionné la similitude de syntaxe entre les messages de communication (Comm) et les scripts, en précisant qu'une communication est en fait un script d'une seule ligne. De plus nous avons vu d'une part que la définition d'un message Comm est particulièrement facile grâce aux constructeurs de communication, et d'autre part que la syntaxe des scripts est un peu compliquée.

Il est possible qu'un programme aie besoin de construire lui-même une chaîne de caractères qui sera utilisée comme un script. La construire "à la main" à l'aide des fonctions sur les chaînes de caractères ne sera pas facile, notamment si le script contient des arguments chaînes de caractères incluant des caractères spéciaux tels que " et \, ou si le script contient des arguments entiers à écrire en hexadécimal.

Pour simplifier cela, le langage SCOL offre une passerelle entre les objets Comm et les chaînes de caractères S ; la fonction suivante permet de convertir une structure Comm en une chaîne utilisable par un script :

`mkscript : fun [ Comm ] S`

La chaîne obtenue contient une ligne de script suivie d'un retour chariot, ce qui permet de concaténer très simplement les lignes de script.

Exemple :

```
defcom loadScr = _load S ;;
defcom mainScr= main S I ;;
...
_openchannel  ":1234"
      strcat (mkscript loadScr ["foo.pkg"] )
            (mkscript mainScr ["bar" 123] )
      _envchannel _channel ;
...
```

Les parenthèses sont ici inutiles.

Lors de l'exécution, le deuxième argument de la fonction `_openchannel` vaudra :

```
"_load \"foo.pkg\"\nmain \"bar\" 7B"
```

4.3. Méthodologie de la programmation réseau en SCOL

Le principe de la communication en SCOL est donc le suivant : les machines s'échangent des messages sur des canaux. Lorsqu'une machine reçoit un message sur un canal, elle regarde si ce message est de nature à déclencher un traitement : pour cela il faut et il suffit qu'il existe une fonction correspondant au message, avec le bon nombre et le bon type d'arguments.

Un des axiomes de la technologie SCOL est qu'il n'est pas possible de s'assurer de l'intégrité du correspondant. Cela signifie concrètement que si A envoie à B un message X sensé recevoir une réponse Y, rien n'assure que la réponse Y sera effectivement envoyée. La seule chose dont A soit sûr, c'est la manière dont lui traite les messages qu'il reçoit. Dans ce sens, la communication en SCOL prend un caractère humain : lorsque vous parlez, vous n'êtes jamais sûr d'être compris, quels que soient les efforts que vous fournissez. On peut objecter que souvent les machines A et B sont programmées par la même personne et que, dans ce cas, le développeur sait bien que B répond un message Y lorsqu'il reçoit un message X. Cependant, même dans ce cas, il faut intégrer cet axiome d'incertitude car vous vous prémunirez alors :

- de vos propres erreurs. Dans l'exemple précédent, si vous programmez vous-même A **et** B, de telle sorte que A envoie le message X et B retourne alors le message Y, il est parfaitement possible que vous commettiez une erreur et que B ne retourne rien. Ceci est d'autant plus possible que, les communications étant des effets de bord, elles ne peuvent être vérifiées statiquement.
- des malveillances. Un des correspondants A ou B est peut-être modifié de manière malveillante. Dans ce cas, A peut recevoir un message Y en "réponse" à un message X qu'il n'a pas envoyé. De même, B peut recevoir un message X à un moment qui n'était pas celui prévu par le développeur.

Pour réaliser une application réseau, il est intéressant d'appliquer la méthodologie suivante :

1. écrire le schéma des communications, c'est-à-dire écrire la liste des messages échangés entre les correspondants. Préciser la fonction, le nombre et le type des arguments. Ce schéma doit clairement représenter le dialogue des machines, comme on le ferait pour une pièce de théâtre.

2. spécifier le comportement que la machine doit avoir lorsqu'elle reçoit un message. Cette spécification ne doit pas présupposer à quel moment le message est reçu : elle doit couvrir tous les cas, même si cela semble inutile, mais c'est toujours pour la même raison : un message peut arriver n'importe quand, que ce soit par dysfonctionnement ou par malveillance.

Bien souvent, ce comportement spécifiera d'envoyer un ou plusieurs messages que l'on peut considérer abusivement comme une réponse.

3. écrire la liste des `defcom`

4. écrire les fonctions en SCOL qui codent ce comportement. D'une certaine manière, on inverse la situation en semblant considérer le programme SCOL comme un effet de bord du message reçu.

Dans le fichier de log, on voit passer, précédés par le mot "`exec`", tous les messages reçus. Lors d'une erreur, le fichier indique que le message ne peut être interprété soit parce qu'il ne correspond à aucune fonction, soit parce que le nombre ou le type des arguments est incorrect. Dans ce cas, le message est simplement ignoré.

5. GESTION DES FICHIERS

La machine SCOL dispose d'un accès restreint aux fichiers de votre ordinateur. Cet accès se fait en définissant un ou plusieurs répertoires appelés **partitions SCOL**. La machine SCOL ne pourra accéder qu'aux fichiers situés dans un de ces répertoires ou bien dans l'un de leurs sous-répertoires. A un instant donné, une seule des partitions SCOL sera accessible en écriture.

5.1. Partitions SCOL

Une partition SCOL est simplement un répertoire du disque auquel la machine SCOL pourra accéder. Elle pourra également accéder à tous ses sous-répertoires.

On peut définir plusieurs partitions : il est recommandé d'en définir au moins 2, car la première joue un rôle spécial. Typiquement :

- la première partition joue le rôle d'une partition cache : lorsque l'utilisateur visite un site, c'est dans cette partition que seront stockés les fichiers téléchargés depuis ce site. On peut définir un quota pour cette partition : le système vérifiera que le volume des fichiers présents dans cette partition ne dépasse pas le quota.
- la seconde partition est votre partition de travail : c'est dans cette partition que se trouvent vos outils et vos propres réalisations. Vos outils peuvent écrire dans cette partition
- les partitions suivantes sont des caches supplémentaires.

Le principe de fonctionnement des partitions SCOL est le suivant :

- lorsque la machine recherche un fichier, elle le cherche dans la première partition, puis dans la seconde, puis dans la troisième. La recherche s'arrête dès que le fichier est trouvé. Un fichier peut donc masquer un autre fichier de même nom, situé dans une partition postérieure.
- lorsque la machine écrit un fichier, elle le fait dans la première partition
- au démarrage de la machine, la première partition est ignorée : la partition de cache est désactivée. Cela signifie que la recherche des fichiers commence à la deuxième partition, et que l'écriture d'un fichier se fait systématiquement dans la seconde partition
- il est possible à tout moment d'activer la partition cache : à partir de ce moment, la recherche des fichiers commence à la première partition, et l'écriture d'un fichier se fait systématiquement dans la première partition
- il n'est pas possible de désactiver la partition cache une fois que celle-ci a été activée.

On comprend ainsi le mécanisme de sécurité sous-jacent : chaque fois que l'utilisateur se connecte sur un site, on activera la partition cache : toutes les écritures se feront dans cette partition. Cette opération étant sans retour, le site ne pourra pas désactiver la partition cache et écrire dans les autres partitions.

Les partitions de la machine SCOL sont également définies dans le fichier **usm.ini** situé dans le répertoire SCOL (typiquement *C:/Program Files/SCOL*). C'est un fichier texte dont les lignes commençant par '*disk*' définissent chacune une partition : un chemin, suivi éventuellement d'un nombre en décimal indiquant le quota. Comme on vient de le voir, l'ordre est important : c'est la première partition définie qui sera la partition cache

```
# extrait exemple de fichier usm.ini

disk ./Cache 32768
disk ./Partition 0
```

```
disk c:/cdrom
```

Le chiffre 0 en face de la partition `/partition/` indique que la partition de travail est accessible en écriture.

La fonction qui permet d'activer le cache est :

```
_cacheActivate : fun [ ] I
```

Il est possible de modifier dynamiquement le chemin d'une partition. Ceci ne peut se faire que d'une manière bien précise, en **allongeant** le chemin de la partition. Par exemple, si la partition était *C:\SCOL\A*, il est possible de la modifier en *C:\SCOL\A\1*, mais il n'est pas possible d'obtenir *C:\SCOL\B*. La fonction à utiliser est la fonction **`_refine`**. Elle se contente d'allonger le chemin de la première partition (en ajoutant éventuellement des '/' manquants). Ce suffixe ne doit pas comporter certaines séquences de caractères (`'..',' ~',''//', ...`).

```
_refine : fun [S] S
```

L'intérêt de cette fonction est de pouvoir gérer un système multi-utilisateurs. Plusieurs machines SCOL peuvent ainsi fonctionner simultanément dans des partitions différentes, en étant reliées au même SCOL Voy@ger (voir plus loin la partie relative au SCOL Voy@ger), il suffit que le SCOL Voy@ger lance les machines des utilisateurs en insérant au début du script de démarrage de ces machines une fonction `_refine` avec en argument le sous-répertoire dédié à l'utilisateur.

5.2. Types de fichiers

SCOL offre deux types de fichiers : les fichiers **normaux** et les fichiers **signés**.

5.2.1. Fichiers normaux

Dans ce mode, l'utilisateur donne le nom du fichier, sans restriction. En lecture, le fichier est recherché dans les différentes partition, alors qu'en écriture, le fichier est placé dans la partition accessible en écriture (partition cache ou partition de travail). Ce mode offre un niveau de sécurité faible : un fichier est accessible en lecture et en écriture dès lors que l'on connaît son nom. Les caractères utilisables pour écrire le nom d'un fichier sont les caractères alpha-numériques, le point, le underscore, l'espace, le tilde, le tiret et le slash. Le système impose, pour des raisons de sécurité, les restrictions suivantes :

- il n'y a pas deux points consécutifs,
- il n'y a pas deux slash consécutifs,
- le nom ne commence pas par un slash,
- le nom ne commence pas par tilde,
- un tilde ne suit pas un slash.

5.2.2. Fichiers signés

5.2.2.1. Signature par fonction de hachage

Dans ce mode, l'utilisateur donne un *nom en clair*, comme défini ci-dessus, auquel le système appose une signature de type cryptographique sur le contenu du fichier. Cette signature commence par un caractère spécial déterminant le type de signature (dans la première signature implémentée, il s'agit du caractère '#'), puis continue et s'achève par une suite de caractères alphanumériques.

Il est donc pratiquement impossible de deviner le nom d'un tel fichier, si on n'en connaît pas le contenu. En outre il est impossible de modifier le contenu d'un fichier de ce type : toute modification

entraîne un changement de signature et donc un changement de nom. Le niveau de sécurité offert par les fichiers signés est donc élevé.

L'autre intérêt de la signature est de déterminer rapidement et avec exactitude si un fichier a déjà été chargé ; typiquement, lorsqu'un utilisateur contacte un serveur SCOL, celui-ci lui indique la liste des packages dont il aura besoin. En utilisant la signature, l'utilisateur retrouve avec précision les packages dont il dispose déjà dans ses partitions. En particulier, le problème des mise-à-jour est résolu automatiquement.

5.2.2.2. Signature par cookies

Il est possible de signer un fichier avec un mot quelconque : celui-ci sera séparé du nom en clair par le caractère ' ;'. Ce mot sera appelé mot de 'cookies'. Une machine SCOL peut définir un mot de 'cookies', une seule fois, en utilisant la fonction suivante :

```
_setCookies : fun [S] S
```

Etant donné qu'il n'est pas possible de redéfinir un mot de 'cookies', ceci permet de protéger certains fichiers. Typiquement, une machine utilisée en client standard utilisera l'adresse Ip du serveur comme mot de cookies, avant même de compiler les paquets indiqués par le serveur. Les fichiers signés avec ce cookies ne seront pas accessibles par les clients d'autres serveurs.

Pour des raisons de sécurité, lorsqu'une machine SCOL ayant activé son cache lance une autre machine SCOL, cette nouvelle machine :

- active elle-même son cache automatiquement au démarrage,
- définit le cookies '__'

5.3. API de gestion des fichiers

Pour utiliser les fichiers, le type `P` a été créé (P pour *Path*). Il reste globalement opaque à l'utilisateur. Pour des manipulations spéciales en écriture, on introduit le type `W`.

Pour lire un fichier il suffit de connaître son nom relatif aux partitions SCOL (chaîne de caractère de type `S`). La fonction `_checkpack` permet de chercher ce fichier dans les différentes partitions et de retourner un objet de type P qui contient en fait le chemin exact du fichier. Ensuite, il est possible d'utiliser cet objet de type P pour les opérations de lecture. Seule exception, la fonction `_load` qui permet de compiler un package fait elle-même les deux opérations (recherche du fichier et lecture).

Pour écrire un fichier, il suffit de connaître son nom (chaîne de caractère de type S). Soit on dispose déjà du contenu du fichier à écrire (dans une chaîne de caractères) et on utilise la fonction `_storepack`. Soit on souhaite écrire le fichier en plusieurs étapes (streaming), et on utilise la fonction `_getmodifypack` pour obtenir un objet de type `W` qui sera utilisé par les fonctions `_createpack` et `_appendpack`.

SCOL n'offre aucun moyen de connaître le chemin complet d'un fichier, ce serait en effet un problème de sécurité que de le permettre.

```
_checkpack : fun [S] P
```
recherche un package dans les partitions, à partir du nom complet (nom en clair suivi éventuellement d'une signature). retourne `nil` si introuvable.

```
_getpack : fun [P] S
```
charge un fichier dans une chaîne de caractères. Pour obtenir le chemin du fichier, il faut au préalable avoir calculé un `_checkpack`. Si le chemin vaut `nil`, le résultat est aussi `nil`.

```
_PtoSCOL : fun [P] S
```

retourne le nom SCOL complet (relatif à la partition) d'un fichier : c'est l'opération inverse de `_checkpack`. Retourne `nil` si le fichier n'est pas dans une des partitions.

`_GetFileNameFromP : fun [P] S`
retourne le nom seul du fichier, privé du chemin.

`_GetFileNameFromW : fun [W] S`
retourne le nom seul du fichier, privé du chemin.

`_storepack : fun [S S] I`
sauvegarde le premier argument avec comme nom le deuxième. Dans le cas où ce nom comporte une signature, celle-ci est vérifiée. Le résultat est 0 en cas de succès. Les sous-répertoires contenus dans le nom sont automatiquement créés si nécessaire.

`_getmodifypack : fun [S] W`
ressemble à la fonction `_checkpack`. Cependant elle prépare à une écriture avec les deux fonctions suivantes. En particulier, le fichier à créer ou à modifier ne peut être signé par son contenu.

`_createpack : fun [S W]  I`
ouvre le fichier W en écriture, en le réinitialisant et en écrivant la chaîne S.
retourne 0 si succès, -1 si erreur.

`_appendpack : fun [S W]  I`
ouvre le fichier W en écriture, et ajoute en fin de fichier la chaîne S.
retourne 0 si succès, -1 si erreur.

`_WtoP : fun [W] P`
convertit le type W en type P. L'inverse n'est pas possible, pour des raisons de sécurité.

`_load : fun [S] I`
cette fonction a déjà été décrite, elle charge et compile un fichier contenant du code SCOL. En fait le fichier est recherché dans les partitions SCOL de la machine (la fonction `_load` appelle elle-même la fonction `_checkpack`)

`_getlongname : fun [S1 S2 S3] S`
calcule le nom complet d'un fichier :

- S1 est le fichier à sauvegarder
- S2 est le nom en clair
- S3 code le type de signature, en reprenant le caractère spécifique
- "" : pas de signature
- "#" : premier type de signature par le contenu
- ";" : signature de type cookies

Cette fonction retourne le nom complet (nom en clair suivi éventuellement de la signature).

Cette fonction a déjà été décrite dans la librairie standard relative aux chaînes de caractères.

```
/* Exemple de manipulation de fichiers */
/* lit un fichier foo.pkg et le recopie sous le nom 'bar#signature',
   en vérifiant le résultat de chaque étape */

fun main()=
 let _checkpack "foo.pkg" -> path
 in if path==nil then -1
 else let _getpack path -> n
```

```
in if n==nil then -1
else let _getlongname n "bar" "#" -> n2
in if n2==nil then -1
else _storepack n n2;;
```

_listoffiles : fun [S] [S r1]
Retourne la liste des fichiers présents dans un répertoire. Les fichiers sont retournés avec leur nom complet. Cette fonction n'est accessible que si la machine n'a pas encore activé son cache.

_listofsubdir : fun [S] [S r1]
Retourne la liste des sous-répertoires d'un répertoire. Les sous-répertoires sont retournés avec leur nom complet. Cette fonction n'est accessible que si la machine n'a pas encore activé son cache.

5.4. Fonction avancées de lecture de fichier

Il est commode de disposer de fonctions plus fines de lecture de fichier. Pour cela, on définit un type **File** qui correspond à un fichier ouvert en lecture. On obtient un tel objet grâce à la fonction `_FILEOpen` et à partir d'un objet de type **P**. Cet objet est également associé à un canal : il se détruira automatiquement lorsque le canal sera détruit.

File _FILEOpen (channel Chn, file P)
Ouvre un fichier en lecture. Retourne `nil` si erreur.

I _FILEClose (file File)
Ferme un fichier

I _FILESeek (file File, position I, mode I)
Positionne la tête de lecture

I _FILETell (file File)
retourne la position de la tête de lecture

I _FILESize (file File)
retourne la taille du fichier

S _FILERead (file File, size I)
lit un certain nombre de caractères dans le fichier, à partir de la position courante.

5.5. Interface de sélection de fichier

Il est possible d'appeler l'interface graphique de sélection de fichier, en lecture ou en écriture. Reportez-vous pour cela au manuel de référence.

6. PROGRAMMATION EVENEMENTIELLE ET INTERFACES GRAPHIQUES

Nous décrirons ici les principes de base de la programmation événementielle et graphique :

- programmation événementielle : comment gérer les événements que la machine SCOL reçoit (timers, interface homme-machine, …)
- programmation graphique : création d'objets graphiques et plus généralement multimedia

Comme illustration de ces principes nous prendrons deux exemples : celui des fenêtres et celui, non-graphique, des timers.

L'interface graphique de SCOL est l'API la plus volumineuse de la machine. Elle permet de gérer les fenêtres, les zones textes, les boutons, les listes, les menus, les polices de caractères, les bitmaps, ... Elle est détaillée dans le manuel de référence.

6.1. Principes fondamentaux

6.1.1. Canal propriétaire

On appellera **objet** toute ressource "extérieure" au langage SCOL (qu'elle soit graphique ou non) : fenêtre, bouton, bitmap, timer, ...

En SCOL, à chaque objet correspond un canal dit propriétaire. Ainsi, lors de la création d'un objet (fenêtre, bouton, timer, etc...), il faudra spécifier le **canal propriétaire** de l'objet. L'existence de l'objet sera liée à celle du canal. A la fermeture d'un canal, tous les objets associés seront simplement détruits.

Nous avons vu dans la partie Gestion de Fichiers, l'exemple des fichiers de type **`File`**. Pour créer un tel objet, nous avons utilisé la fonction `_FILEOpen` de type `fun [Chn P] File`. On voit ici de quelle manière on a spécifié le canal propriétaire : c'est le premier argument.

De même le type de la fonction _CRwindow rencontré dans les exemples 'Hello World' est :

```
_CRwindow : fun [Chn ObjWin I I I I I S] ObjWin
```

Le type **`ObjWin`** correspond à un objet fenêtre. Pour créer une fenêtre, on spécifie donc le canal propriétaire, suivi de la fenêtre mère, puis de divers arguments fixant la position, la taille, le type et le titre de la fenêtre.

Le rôle du canal propriétaire va plus loin que la simple destruction des objets lorsque le canal tombe. Il s'étend également à la gestion des événements.

6.1.2. Gestion des événements

En effet, la plupart des objets sont susceptibles de recevoir des événements. Par exemple, on peut cliquer sur une fenêtre. On peut déplacer une fenêtre. On peut enfoncer un bouton. Chacune de ces actions correspond à un événement que le programme doit pouvoir traiter. De même, le rôle d'un timer est de provoquer un événement à intervalles réguliers.

6.1.2.1. Différents systèmes de gestion des événements

La question qui se pose est la suivante : comment un programme SCOL reçoit-il les événements ?

C'est une question qui se pose pour tout système d'exploitation, et les réponses sont souvent différentes.

Sous *Windows*, le programme définit une fonction qui est appelée chaque fois qu'un événement est produit par le système, quel que soit l'événement. Sous *Unix*, avec *X-Window* (X11), le principe est similaire, mais l'utilisateur peut choisir quels événements sa fonction doit recevoir. Dans les deux cas, la fonction "trie" les événements et, selon le type d'événement, effectue le traitement approprié. En C, cette fonction se résume le plus souvent à un immense 'switch'.

Avec les *Xt Intrinsics* (modèle haut-niveau de programmation de *X-Window*, utilisé par exemple pour Osf-Motif), le développeur définit une fonction par type d'événement. Une telle fonction ne sera appelée que lorsqu'un événement d'un type précis se produira. On appellera une telle fonction un '**réflexe**' ou encore une '**callback**'. Cette méthode présente deux avantages : d'une part, il n'y a plus à écrire la fonction 'switch', d'autre part, le passage des paramètres de l'événement se fait de manière plus simple. Par exemple, un événement click se caractérise par trois paramètres : les coordonnées du click et le numéro du bouton. Un événement `timer` ne se caractérise par aucun paramètre. Un événement 'le contenu d'une zone texte a changé' se caractérise par un paramètre : le nouveau texte. Faire traiter tous ces événements par une même fonction pose un problème de passage de paramètres : on est souvent (*Windows, X-Windows*) obligé de prendre des libertés avec le typage, ce qui est une grande source d'erreur. L'avantage du système de réflexes est de pouvoir définir un type de fonction réflexe par type d'événement. Comme vous l'aurez deviné, c'est cette manière de gérer les événements qui a été retenue pour SCOL.

6.1.2.2. Définition des fonctions réflexe

En SCOL, on pourra donc définir pour chaque objet et chaque type d'événement une fonction réflexe. Cette fonction réflexe devra prendre au moins deux arguments :

- le premier est l'objet affecté
- le second est un paramètre utilisateur quelconque

Selon les événements, la fonction réflexe aura des arguments supplémentaires : 3 entiers pour un click, deux entiers pour un événement 'redimensionnement de la fenêtre', pas d'argument supplémentaire pour un événement timer.

Pour définir une fonction réflexe, on utilisera la fonction de définition appropriée. Par exemple, dans le programme 'hello3' décrit dans la partie 'Hello World', on trouve la ligne suivante :

```
_CBwinDestroy win @_end nil;
```

La fonction `_CBwinDestroy` permet de définir le réflexe associé à l'événement 'destruction de la fenêtre'. Elle prend trois arguments :

- la fenêtre dont on définit le réflexe (ici '`win`')
- la fonction réflexe (ici '`@_end`')
- le paramètre utilisateur (ici '`nil`')

Le type de la fonction réflexe est : `fun [ObjWin ?] ?`, le deuxième argument étant le paramètre utilisateur. Le type du résultat est indifférent. Le type de la fonction `_CBwinDestroy` est donc :

_CBwinDestroy : fun [ObjWin fun [ObjWin u0] u1 u0] ObjWin

On remarque comment le typage assure que le paramètre utilisateur fourni lors de la définition du réflexe a le même type que le paramètre utilisateur de la fonction réflexe (ici '`u0`'). SCOL apporte ici un plus par rapport à Xt-Intrinsics : le paramètre utilisateur des Xt-Intrinsics n'est pas typé : il oblige le développeur à jouer avec les types, ce qui est toujours une grande source d'erreur.

Autres exemples :

Un événement click sur une fenêtre nécessite trois paramètres : coordonnées du click et numéro du bouton. La fonction réflexe associée est de type : `fun [ ObjWin u0 I I I] ?`

La fonction de définition de la fonction réflexe click est :

```
_CBwinClick : fun [ObjWin fun [ObjWin u0 I I I] u1 u0] ObjWin
```

Un timer est un objet de type **`Timer`**.

La fonction réflexe associée au timer est de type : `fun [ Timer u0] ?`

La fonction de définition de la fonction réflexe timer est :

```
_rfltimer : fun [Timer fun [Timer u0] u1 u0] Timer
```

Si on définit un réflexe avec **`nil`** comme deuxième argument (en lieu et place de la fonction réflexe), cela supprime le réflexe. De même, on peut redéfinir un réflexe à tout moment, y compris dans la fonction réflexe elle-même.

6.1.2.3. Traitement d'un événement

Lorsqu'un événement se produit, la machine SCOL recherche l'objet concerné par l'événement, puis la fonction réflexe associée à cet événement. Si celle-ci a été définie, la machine SCOL exécute cette fonction et oublie le résultat, sauf dans certains cas bien particuliers où le résultat est interprété par le système. Cette fonction doit s'exécuter dans un canal (en effet, cette fonction appelle peut-être les fonctions `_load` ou `_closechannel`). C'est le **canal propriétaire** qui est sélectionné : nous voyons là le second rôle du canal propriétaire.

On peut changer le canal propriétaire d'un objet ; plus exactement, il est possible d'attribuer tous les objets d'un canal à un autre canal grâce à la fonction suivante :

`_chgchn : fun [Chn`$_1$ **`Chn`**$_2$**`] Chn`**$_2$

Attribue tous les objets du canal Chn_1 au canal Chn_2.

Cette fonction permet en particulier de sauver de la destruction les objets d'un canal juste avant que celui-ci ne soit détruit (lors de l'événement `_closed` par exemple).

6.2. Exemples

6.2.1. Fenêtres

Nous illustrons ici aux fenêtres les principes qui viennent d'être exposés. La documentation complète des fonctions suivantes se trouve dans le manuel de référence.

```
ObjWin _Crwindow( Chn channel, ObjWin parent, I posx, I posy,
 I taillew, I tailleh, I flags, S name)
```

Cette fonction crée une nouvelle fenêtre.

```
I _DSwindow ( ObjWin fenetre )
```

Cette fonction détruit une fenêtre.

```
ObjWin _CBwinPaint ( Objwin fenetre, fun [ ObjWin u0 ] u1funreflex, u0
param )
```

réflexe de l'événement '*Paint*'. Pas d'argument particulier.

```
ObjWin _CBwinMove ( ObjWin fenetre, fun [ ObjWin u0 I I ] u1funreflex, u0
param)
```

réflexe de l'événement '*Move*'. Deux arguments qui donnent la nouvelle position.

`ObjWin _CBwinSize ( ObjWin fenetre, fun [ ObjWin u0 I I ] u1 funreflex, u0 param)`
réflexe de l'événement '*Size*'. Deux arguments qui donnent la nouvelle taille.

`ObjWin _CBwinClick ( ObjWin fenetre, fun [ ObjWin u0 I I I] u1 funreflex, u0 param)`
réflexe de l'événement '*Click*'. Trois arguments qui donnent les coordonnées du click et le bouton de la souris utilisé.

`ObjWin _CBwinUnclick ( Objwin fenetre, fun [ ObjWin u0 I I ] u1, funreflex, u0 param)`
réflexe de l'événement '*UnClick*'. Trois arguments qui donnent les coordonnées du '*unclick*' et le bouton de la souris utilisé.

`ObjWin _CBcursorMove ( ObjWin fenetre, fun [ ObjWin u0 I I I] u1 funreflex, u0 param)`
réflexe de l'événement '*CursorMove*'. Trois arguments qui donnent les coordonnées de la souris et le bouton de la souris utilisé.

`ObjWin _CBwinKeydown ( ObjWin fenetre , fun [ ObjWin u0 I I ] u1 funreflex, u0 param)`
réflexe de l'événement '*KeyDown*'. Deux arguments qui donnent le '*scancode*' de la touche enfoncée ainsi que la valeur de la touche enfoncée. Voir les annexes pour les valeurs spéciales de cet argument.

`ObjWin _CBwinKeyup ( ObjWin fenetre, fun [ ObjWin u0 I ] u1funreflex, u0 param)`
réflexe de l'événement '*KeyDown*'. Un argument qui donne le '*scancode*' de la touche relevée.

`ObjWin _CBwinDestroy ( ObjWin fenetre, fun [ ObjWin u0 ] u1funreflex, u0 param)`
réflexe de l'événement '*Destroy*'. Pas d'argument particulier.

`ObjWin _CBwinFocus ( ObjWin fenetre , fun [ ObjWin u0 ] u1 funreflex, u0 param )`
réflexe de l'événement '*Focus*'. Pas d'argument particulier.

`ObjWin _CBwinKillFocus ( ObjWin fenetre , fun [ ObjWin u0 ] u1 funreflex, u0 param )`
réflexe de l'événement '*Perte de Focus*'. Pas d'argument particulier.

`ObjWin _CBwinDClick (ObjWin fenetre,fun [ObjWin u0 I I I] u1 , u0 param )`
réflexe de l'événement '*DoubleClick*'. Trois arguments qui donnent les coordonnées du click et le bouton de la souris utilisé.

`ObjWin _CBwinDropFile ( ObjWin fenetre, fun [ObjWin u0 I I [P r1]]u1 u0)`
réflexe de l'événement '*DropFile*'. Trois arguments qui donnent les coordonnées du click et la liste des fichiers "déposés" sur la fenêtre par l'utilisateur.

6.2.2. *Timers*

Il est possible de définir des timers, dont la période est définie en millisecondes. Un timer de période 1000 provoquera un événement timer toutes les secondes. Les objets '*timers*' sont de type '**`Timer`**'. Trois fonctions permettent de gérer les timers :

`_starttimer : fun [Chn I] Timer`
Crée un timer en définissant le canal propriétaire, avec une certaine période exprimée en millisecondes. Retourne l'objet '*timer*', '`nil`' si erreur.

`_deltimer : fun [Timer] I`
Détruit le timer passé en argument.

`_rfltimer : fun [Timer fun [Timer u0] u1 u0] Timer`
définit de la fonction réflexe associée au timer.

7. PROGRAMMATION 3D

La machine SCOL contient une librairie capable de traiter et d'afficher des scènes en 3 dimensions. On appelle cette librairie un '**moteur 3d**'. Cette librairie permet très simplement d'ajouter des fonctionnalités 3d à vos programmes. C'est l'un des intérêts du langage SCOL de combiner finement des capacités de communication internet et des fonctionnalités 3d puissantes.

Le moteur 3d de SCOL a été spécialement étudié pour une utilisation on-line : les fichiers de description des scènes ne sont pas volumineux, et le rendu fonctionne de manière fluide sur des machines peu puissantes. En effet, il est a noté que la population qui utilise internet n'est pas toujours équipée de la machine dernier-cri, contrairement à la population qui achète des jeux-vidéos. De même, il n'est pas possible sur internet d'obliger les utilisateurs à s'équiper de carte 3d, ni même à s'assurer qu'ils ont le dernier driver pour leur carte.

Pour pouvoir comprendre ce chapitre, vous devrez être un peu familier des notions liées à la 3d (scènes, polygones, matériaux, textures, rendu, ...), même si la plupart des termes seront ré-expliqués. De plus, pour bien comprendre les exemples, vous devez avoir au moins parcouru la documentation de SCOL concernant les interfaces graphiques : il est nécessaire pour afficher une image 3d de créer une fenêtre, un objet surface et de copier l'objet surface dans la fenêtre. Trois fonctions suffiront dans un premier temps.

On divisera le chapitre en plusieurs parties. Dans la première, on rappellera les notions 3d de base. Dans la seconde, on détaillera les fichiers de définition de scène. Dans la troisième, on donnera les fonctions de manipulation et de rendu. Dans la quatrième, on abordera le problème délicat des collisions.

7.1. Notions de base pour la 3d

7.1.1. Scène

La notion de scène est à la base de la 3d. Une scène est un ensemble d'éléments tels que des objets 3d, des caméras ou des objets de collision. Cet ensemble n'est pas quelconque : il s'agit en fait d'une arborescence. Chaque noeud de l'arbre est un objet 3d, une caméra, ou un objet de collision. Le nœud qui n'a pas de père s'appelle la **racine** de l'arbre. On définit logiquement les notions de **nœud père**, de **nœud fils** et de **nœud frère**.

D'un point de vu géométrique, chaque nœud définit un **repère**. Le repère de la racine sera dit **repère global**. Un repère est caractérisé par rapport à celui du nœud père par des **coordonnées** de **position** et d'**orientation**, ainsi qu'un paramètre d'échelle.

Les objets 3d seront ici des ensembles de polygones. Les polygones seront soit des triangles, soit des quadrilatères convexes. Par exemple, un cube sera formé de 6 polygones : un carré pour chaque face du cube. Sur chaque polygone (ou face), on pourra appliquer soit une couleur uniforme sur la face, soit une texture, à savoir une image quelconque. Les objets 3d seront appelés '**mesh**'.

Il existe un cas particulier des objets 3d : c'est l'objet 3d vide, qui ne contient aucun polygone. Cet objet sera appelé '**shell**'. Un 'shell' permet donc de définir un repère vide auquel on pourra lier différents éléments. En déplaçant ce 'shell', on déplacera tous les éléments attachés.

Les caméras ressemblent à de vraies caméras : on définit notamment une focale. Les caméras seront appelées '**camera**'.

Les objets de collisions sont des ensembles de sphères, de parallélépipèdes rectangles ou de triangles qui définissent des zones pleines de l'espace. Ces objets seront appelés '**coll**'.

Le finalité principale du moteur 3d est de calculer l'image que "voit" une certaine caméra. Pour cela, il suffit de spécifier une caméra et une surface dans lequel l'image sera calculée. La scène ainsi prise en compte sera celle à laquelle la caméra appartient.

7.1.2. Notion de session.

Le moteur 3d permet de gérer plusieurs scènes simultanément. En fait, la notion de scène est remplacée par celle de session. Une session 3d est un ensemble d'éléments tels que ceux décrits précédemment (mesh, shell, caméra, coll). Chaque élément possède éventuellement un père ; il y a autant de scènes que d'éléments orphelins.

7.1.3. Matériau

Un matériau est un élément qui détermine l'apparence d'un polygone.

Un matériau peut être '**flat**' : cela signifie qu'une couleur uniforme est appliquée sur la face. Cette couleur peut varier selon l'éclairage, c'est-à-dire selon l'orientation de la face avec les rayons de lumière. Cette couleur peut également être translucide : elle se comporte comme un filtre de couleur. La puissance du filtre varie entre la transparence totale, et l'opacité pure et simple.

Un matériau peut être '**texturé**' : cela signifie qu'une image est appliquée sur la face. Cette image peut être un dessin ou une photo. Le matériau peut être transparent : cela signifie que les points d'une certaine couleur (que l'utilisateur peut déterminer) seront considérés comme transparents. On pourra affecter aux autres un coefficient de transparence.

Sur un matériau texturé, on pourra appliqué des filtres divers de colorimétrie : filtre vers une couleur, rotation des couleurs, changement de saturation, ...

Les matériaux sont des éléments de la session. Ce ne sont pas des éléments géométriques : ils ne sont donc pas attachés à des repères, et n'ont bien sûr pas de père.

7.1.4. Performances

En terme de performance, il est toujours difficile d'être précis. On peut donner les principes généraux qui permettent d'optimiser des scènes.

7.1.4.1. Vitesse d'exécution

La vitesse d'exécution concerne surtout la vitesse de rendu. Les autres opérations (déplacements, rotations, ...) sont très rapides.

- moins il y a de polygones, plus le rendu est rapide (éviter de dépasser 15000).
- à nombre de polygones constant, le rendu est d'autant plus rapide qu'il y a beaucoup de meshes dans la scène
- le matériau flat est beaucoup plus rapide que le matériau texturé
- la transparence (pour le flat ou pour le texturé) est particulièrement gourmande, surtout lorsqu'il y a superposition de faces transparentes.

7.1.4.2. Place en mémoire

La place utilisée dans la mémoire de votre ordinateur est un paramètre important : si la scène est trop volumineuse, votre ordinateur manquera de mémoire et utilisera les mécanismes de *'swap'*, très

gourmands en temps. Il est important de distinguer la taille de vos fichiers 3d et la place en mémoire utilisée pour les stocker. Par exemple un fichier contenant une image au format '*jpeg*', de résolution 256x256, pourra avoir une taille de 6Ko (s'il est suffisamment compressé), mais occupera toujours 128Ko en mémoire. C'est ce dernier chiffre qui est important ; il est obtenu de la manière suivante : 256x256x2, car il faut deux octets pour stocker la couleur d'un point.

La taille occupée par les informations autres que les textures n'est pas vraiment significative : 2Mo permettra normalement de gérer des scènes de 15000 polygones.

Au-delà de 10 Mo de textures, votre scène tournera difficilement sur un ordinateur ayant 32Mo de mémoire vive. Soyez donc raisonnables quant au poids des textures.

7.1.5. Caractéristiques du moteur 3d SCOL.

7.1.5.1. Gestion mémoire

Le moteur utilise une bande de mémoire dédiée, qu'il utilise pour stocker toutes ses données, exception faite des textures. Cette bande mémoire sert également à générer le rendu.

Le moteur gère une liste de matériaux, ainsi qu'une liste de textures. Le ménage est fait automatiquement, grâce à un GC (Garbage Collector) de type "compteur de références"

La gestion de la mémoire est automatique pour les points suivants :

- lorsqu'un matériau n'est plus référencé par un objet, il est supprimé
- lorsqu'une texture n'est plus référencée par un matériau, elle est supprimée

De cette manière, il n'est pas possible de copier un matériau sans spécifier au moins un objet qui utilise la copie. De même, il n'est pas possible de copier une texture sans spécifier au moins un matériau qui utilise la copie.

On a donc trois types de données dans la bande mémoire :

- les objets qui référencent un nombre quelconque de matériau (au plus un par polygone)
- les matériaux qui référencent au plus une texture
- les textures qui sont soit chargées (l'image de la texture est en mémoire), soit déchargées (l'image de la texture n'est pas encore en mémoire). Dans ce dernier cas, les matériaux qui référencent ces textures sont automatiquement rendus en flat.

7.1.5.2. Scène

Les objets de la scène sont organisés sous forme d'arbres. Les objets sont de quatre types :

- type " shell " : objet vide
- type " mesh " : ensemble de points et de polygones
- type " camera " : caméra de visualisation
- type " coll " : objet de collision

Les fonctions de déplacements, de création, de suppression, s'appliquent sur les objets en général, sans distinction de type, ce qui simplifie l'API.

7.1.5.3. Matériaux

Il y a deux types généraux de matériaux : les matériaux texturés et les matériaux non texturés. Tout matériau dispose d'un mode non texturé. Il est possible de charger les textures à tout moment.

Le rendu se fait en 15 bits non paletté. Les textures sont au format **bmp** 8 bits paletté (aujourd'hui nettement déconseillé car trop lourd à télécharger, et de qualité médiocre) ou au format **jpeg**. Il est recommandé de n'utiliser que du jpeg, car c'est le format qui offre les meilleures performances de compression.

Les matériaux texturés disposent d'un mode transparent. Pour les textures au format bmp, la couleur 0 est la couleur de transparence. Pour les textures au format jpeg, la couleur de transparence peut être spécifiée.

Le texturage peut se faire en 'mapping environnemental' (qui permet des effets de réflexion, notamment pour obtenir des effets métalliques)

Différents filtres peuvent être appliqués :

- filtre de couleur : calcule pour chaque point le barycentre entre la couleur du point et la couleur du filtre. Le coefficient du barycentre est variable
- filtre attracteur : un point dont la couleur est située à une distance de la couleur du filtre inférieure à une certaine valeur est forcé à la couleur du filtre
- filtre de saturation : calcule pour chaque point le barycentre entre la saturation du point et la saturation du filtre. Le coefficient du barycentre est variable
- filtre de rotation : fait tourner d'un certain angle la teinte d'une couleur.

Les matériaux peuvent utiliser un niveau de transparence, variant entre 0 (opaque) et 255 (transparent). Dans le cas du moteur **software** (n'utilisant pas de carte 3d), les matériaux texturés ne peuvent avoir que deux niveaux de transparence :

- de 0 à 127 : opaque
- de 128 à 255 : transparence 50%

Le lissage Gouraud est disponible en mode software et en mode hardware.

7.2. Format des fichiers 3d

Pour définir une scène, il faut définir différents éléments : meshes, caméra, objets de collision, matériaux, ... Il est pratique de posséder un format de fichier qui permet de décrire simplement et complètement une scène, ou bien une partie seulement d'une scène. On verra que le moteur 3d offre des fonctions de manipulation qui permettent par la suite de modifier pratiquement chaque caractéristique définie par les fichiers de scène. Le format des fichiers 3d de SCOL s'appelle M3D.

Le format m3d permet de décrire des scènes hiérarchisées contenant des matériaux, des meshes, des caméras, et d'autres fichiers à inclure. Un fichier m3d est un fichier texte que l'on peut donc ouvrir avec n'importe quel éditeur. L'élément de base du fichier est la ligne. Le fichier se divise en blocs de type :

```
type_d'élément nom_de_l'élément {
... (contenu)
}
```

Le type d'élément vaut : material, mesh, shell, camera, coll.

Les blocs peuvent s'imbriquer (sauf les blocs 'material') :

```
shell x {
... (contenu)
mesh y {
... (contenu)
}
mesh z {
... (contenu)
camera w {
```

```
... (contenu)
}
}
}
```

Cette imbrication permet de rendre compte de la structure arborescente d'une scène.

Les blocs matériaux sont de la forme :

```
material nom_de_materiau {
color 123456
texture nom_de_texture.jpg
type NOLIGHT
type TRANSPARENCY120
}
```

Aucune des lignes n'est ici indispensable. Cependant, il va de soi qu'on doit définir au minimum une couleur de flat (ligne '`color`') ou un fichier de texture (ligne '`texture`').

La couleur est donnée en hexadécimal 6 digits RGB (8bits R, 8bits G, 8bits B). Par exemple, le rouge vif est codé ff0000.

La texture est un nom de fichier qui peut être précédé d'un filtre, placé entre '%'. Le filtre est une chaîne de caractères qui constitue une liste de traitements :

- C[6 caractères de couleur][2 caractères de taux] : filtre de couleur (exprimée en 24bits hexa) de taux variant entre 0 et 255 (2 caractères hexa)
- X : échange les couleurs Rouge->Bleu->Vert->Rouge
- Y : échange les couleurs Rouge->Vert->Bleu->Rouge
- A[6 caractères de couleur][2 caractères de distance] : attracteur. Tous les points proches de la couleur exprimée sont forcés à cette couleur. Cette couleur devient la couleur de transparence pour les textures jpeg
- R[4 caractères de rotation] : rotation de teinte (modèle Hsv). L'angle est compris entre 0 et 65535.
- S[4 caractère de valeur][2 caractères de taux] : filtre de saturation. La valeur limite de la saturation est donnée sur 4 octets, et vaut entre 0 et 65536. Le taux varie entre 0 et 255

Exemple : `%Cd0000080X%toto.jpg`

Dans cet exemple, on appliquera à la texture *toto.jpg* un filtre rouge à 50% suivi d'une rotation de couleurs X.

Les lignes de types sont : NOLIGHT, TRANSPARENCY120. On peut mettre autant de lignes type qu'on le souhaite. Le coefficient qui suit le mot TRANSPARENCY vaut entre 0 (opaque) jusqu'à 255 (transparent). Il n'est utile que lors d'un rendu non texturé.

Les blocs mesh sont de la forme :

```
mesh nom_du_mesh {
x y z a b c scale
vertices
[light l]
polygones
[récursivité]
}
```

La position du mesh est donnée par les coordonnées `x, y, z` et les angles `a, b, c` (angles compris entre 0 et 65535). Le paramètre `scale` est optionnel. Il est exprimé en pourcentage : 100 représente l'échelle normale. 200 pour une taille double, 50 pour une demi-taille, ...

Les vertices sont des lignes de trois coordonnées `x, y, z` entières ou flottantes.

La lumière `l` est facultative, et doit être comprise entre 0 (très sombre) et 31 (valeur utilisée par défaut).

Les polygones sont décrits sous forme de blocs comprenant :

- une ligne donnant un nom de matériau à appliquer sur les polygones à suivre
- des lignes de polygones comprenant 3, 4, 6, 8, 9 ou 12 entiers, selon qu'on définit 3 ou 4 sommets, avec 0, 1 ou 2 coordonnées de textures.

A la suite des polygones, il y a moyen d'insérer d'autres meshes, caméras.

Les blocs caméra sont de la forme :

```
camera nom_de_camera {
x y z a b c scale
distx disty sx sy
zclip zbrouillard zback
}
```

La position de la caméra est donnée par les coordonnées `x, y, z` et les angles `a, b, c` (angles compris entre 0 et 65535). Le paramètre `scale` est optionnel.

Les paramètres `distx` et `disty` donnent la distance de l'écran sur l'axe x et sur l'axe y (utilisés dans les formules de projection `X=distx*x/z` et `Y=disty*y/z`).

Les paramètres `sx` et `sy` donnent la demi-largeur et la demi-hauteur de l'écran.

On appelle **Shell** un objet vide (ni caméra, ni mesh) qui ne sert qu'à lui accrocher d'autres objets. La syntaxe d'un objet **Shell** est la suivante :

```
shell nom_de_shell {
x y z a b c scale
}
```

A la suite des coordonnées du **shell**, il y a moyen d'insérer d'autres meshes, caméras.

Le caractère `#` indique que la suite de la ligne est un commentaire.

Exemple :

```
# cube
material wood {
 color c04000
 texture wood.jpg
}

material pierre {
 color 00c040
 texture pierre.jpg
 type TRANSPARENCY
}

mesh cube {
0 0 0  0 0 0

-100 -100 -100
100 -100 -100
100 100 -100
-100 100 -100
-100 -100 100
100 -100 100
100 100 100
-100 100 100

light 10
wood
```

```
 0 0 0  1 255 0  2 255 255 3 0 255
 1 0 0  5 255 0  6 255 255 2 0 255
 5 0 0  4 255 0  7 255 255 6 0 255
pierre
 4 0 0  0 255 3  3 255 255 7 0 255
 3 0 0  2 255 0  6 255 255 7 0 255
 1 0 0  0 255 0  4 255 255 5 0 255
}
```

Dans cet exemple, on a défini un cube d'arête 200. Il y a trois faces qui utilisent le matériau 'wood' et trois faces qui utilisent le matériau 'pierre'.

7.3. API 3d de manipulation

L'exemple de base comporte les points suivants :

- création d'un bitmap dans lequel on effectuera la rendu
- création d'une session
- lecture d'un fichier m3d contenant un cube et une caméra
- calcul du rendu
- affichage du bitmap dans une fenêtre.

Pour cela, on utilisera trois fichiers :

fichier '*Tutorial/mytest3d.SCOL*'

```
_load "Tutorial/mytest3d.pkg"
main
```

fichier '*Tutorial/mytest3d.pkg*'

```
/* MyTest 3d */
typeof win=ObjWin;;
typeof buffer=ObjSurface;;
typeof session=S3d;;
typeof shell=H3d;;
typeof camera=H3d;;

fun _end(a,b)=_closemachine;;
fun _paint(a,b)=_BLTsurface win 0 0 buffer 0 0 400 300;;

fun main()=
 set win=_CRwindow _channel nil 150 150 400 300
  WN_MENU|WN_MINBOX "My 3d Test";
 _CBwinDestroy win @_end nil;
 _CBwinPaint win @_paint nil;
 set buffer=_CRsurface _channel 400 300;
 set session = MX3create _channel 1024 1024 1024 1024 1024*1024;
 if session==nil then _closemachine
 else
 (set shell = M3createShell session;
  M3load session "Tutorial/scene.m3d" shell;
  set camera=M3getObj session "camera";
  M3recursFillMatObj session shell;
  MX3render session buffer camera 0 0 0;
  _paint nil nil;
  0);;
```

fichier '*Tutorial/scene.m3d*'

```
# cube
material wood {
 color c04000
 texture Tutorial/wood.jpg
}

material pierre {
 color 00c040
 texture Tutorial/stone.jpg
}

mesh cube {
0 0 0  0 0 0

-100 -100 -100
100 -100 -100
100 100 -100
-100 100 -100
-100 -100 100
100 -100 100
100 100 100
-100 100 100

wood
 0 0 0  1 255 0  2 255 255 3 0 255
 1 0 0  5 255 0  6 255 255 2 0 255
 5 0 0  4 255 0  7 255 255 6 0 255
pierre
 4 0 0  0 255 3  3 255 255 7 0 255
 3 0 0  2 255 0  6 255 255 7 0 255
 1 0 0  0 255 0  4 255 255 5 0 255

camera camera {
200 300 -400 7000 -5000 0
200 200 200 150
10 10000 10000
}
}
```

Si vous essayez l'exemple, vous obtiendrez une fenêtre contenant un cube vu de trois-quart. Les faces sont colorées. Pour obtenir les textures, vous devez créer deux fichiers graphiques '*Tutorial/stone.jpg*' et '*Tutorial/wood.jpg*'. Vous pouvez également renommer ces fichiers au début du fichier *Tutorial/scene.m3*d, et les remplacer par des fichiers *jpeg*.

Dans l'exemple, nous remarquons quelques fonctions nouvelles, dont le nom commence par '`M3`' ou '`MX3`' : ce sont les fonctions de l'API du moteur 3d de SCOL. Présentons les rapidement avant de passer à la liste exaustive de ces fonctions.

`set session = MX3create _channel 1024 1024 1024 1024 1024*1024;`
Crée une session 3d dont la taille de la bande mémoire est 1Mo. La variable session a le type 'S3d'

`(set shell = M3createShell session;`
Crée un élément shell dans la session. La variable `shell` a le type 'H3d'

`M3load session "Tutorial/scene.m3d" shell;`
Lit le fichier *Tutorial/scene.m3d*. et place tout ce qu'il contient sous l'élément shell.

`set camera=M3getObj session "camera";`

Retrouve la trace de l'objet appelé 'camera' dans la scène. La variable `camera` a le type 'H3d'

`M3recursFillMatObj session shell;`
Charge toutes les textures de la scène, ou plus exactement les textures utilisées par les objets situés sous l'objet shell

`MX3render session buffer camera 0 0 0;`
Calcule le rendu de ce que 'voit' la caméra, dans le buffer.

Note : les fonctions qui commencent par MX3 ont été introduites pour supporter les cartes 3d, et passent en mode soft s'il n'y a pas de carte 3d. Les anciennes fonctions `M3create` et `M3scanline` sont donc désormais obsolètes.

7.3.1. Nouveaux types

On définit les types magma suivants :

- `S3d` session 3d
- `H3d` handler objet 3d
- `Hmat3d` handler matériau
- `Htx3d` handler texture

D'une manière générale, les fonctions retournent 0 en cas de succès.

7.3.2. Session

`S3d MX3create(channel Chn, nbmat I, nbtext I, nbobj I, ntaby I, sizetape I)`
Initialise une session 3d, en spécifiant le nombre maximum de matériaux, de textures, d'objets et de lignes de rendu, ainsi que la taille de la bande mémoire à réserver pour le stockage des meshes et le calcul du rendu.

`I M3destroy(session H3d)`
Désalloue la session 3d.

`I M3freeMemory (session S3d)`
Retourne la taille mémoire encore libre dans la session. Effectue un GC.

`I M3reset (session S3d)`
Efface tous les objets et toutes les textures de la session.

7.3.3. Gestion générale des objets

`I M3load(session S3d, fichier S, père H3d)`
Charge un fichier au format m3d en spécifiant le handler du père (`nil` si aucun). Si le père est spécifié, les éléments du fichier seront considérés comme étant leur fils.

`I M3loadString(session S3d, contenu S, père H3d)`
Charge un objet depuis une chaîne de caractères au format m3d, en spécifiant le handler du père (`nil` si aucun).

`H3d M3createShell(session S3d, père H3d)`
Crée un objet de type Shell, et retourne son handler (`nil` si échec)

`I M3delObj(session S3d, objet H3d)`
Détruit un objet, en connaissant son handler.

H3d M3copyObj(session S3d, objet H3d)
Crée une copie de l'objet. Cette copie est détachée de toute arborescence.

H3d M3getObj(session S3d, nom S)
Donne le handler d'un objet, en fonction de son nom. `nil` si introuvable.

S M3objName(session S3d, objet H3d)
Retourne le nom de l'objet.

H3d M3getFather(session S3d, objet H3d)
Retourne le handler du père d'un objet.

H3d M3getFirstSon(session S3d, objet H3d)
Retourne le handler du premier fils d'un objet.

H3d M3getBrother(session S3d, objet H3d)
Retourne le handler du frère d'un objet.

H3d M3bigFather(session S3d, objet H3d)
Retourne le handler du sommet de l'arbre auquel un objet appartient.

H3d M3isFather(session S3d, fils H3d, père H3d)
Retourne `1` si pere est bien au-dessus de fils dans l'arborescence.

I M3unLink(session S3d, père H3d)
Détache un objet (et toute sa descendance). L'objet reste en mémoire.

I M3link(session S3d, fils H3d, père H3d)
Attache un objet sous un autre.

I M3renameObj(session S3d, objet H3d, name S)
Change le nom d'un objet (le nom doit avoir moins de 32 caractères)

I M3setObjVec(session S3d, objet H3d ,vector [I I I])
Définit la position d'un objet dans le repère de son père

[I I I] M3getObjVec(session S3d, objet H3d)
Lit la position d'un objet dans le repère de son père

I M3setObjAng(session S3d, objet H3d, angular [I I I])
Définit la position angulaire d'un objet.

[I I I] M3getObjAng(session S3d, objet H3d)
Lit la position angulaire d'un objet.

I M3setObjScale(session S3d, objet H3d, scale I)
Définit l'échelle d'un objet, exprimée en pourcentage : 100 pour la taille normale, 200 pour une taille double, ...

I M3getObjScale(session S3d, objet H3d)
Lit l'échelle d'un objet, exprimée en pourcentage : 100 pour la taille normale, 200 pour une taille double, ...

I M3movObj(session S3d, objet H3d, vector [I I I])

Déplace un objet d'un certain vecteur dans le référentiel de l'objet. Pour une caméra, l'axe de vision est z, l'axe horizontal est x, et l'axe vertical est y.

`I M3rotateObj(session S3d, objet H3d, angular [I I I])`
Tourne un objet d'un certain vecteur angulaire dans le référentiel de l'objet.

`I M3movObjExt(session S3d, objet H3d, vector [I I I])`
Déplace un objet d'un certain vecteur dans le référentiel du père de l'objet.

`I M3rotateObjExt(session S3d, objet H3d, angular [I I I])`
Tourne un objet d'un certain vecteur angulaire dans le référentiel du père de l'objet.

`I M3calcMat(session S3d, objet H3d);`
Calcule les positions des objets d'une scène dans le repère de l'objet. On peut récupérer ces positions avec les deux fonctions suivantes.

`[I I I] M3getObjVecRender(session S3d, objet H3d)`
Donne la position d'un objet après `M3calcMat` ou `M3scanline`.

`[[I I I] [I I I] [I I I]] M3getObjMatrixRender(session S3d, objet H3d)`
Donne la matrice d'un objet après `M3calcMat` ou `M3scanline`. La matrice est constituée d'entiers 32bits, avec 16bits après la virgule.

`[[I I] [I I] [I I I]] M3getCamera(session S3d, objet H3d)`
Donne les paramètres d'une caméra : `[ [dx dy] [sx sy] [zclip zmiddle zmax] ]` :

- `dx, dy` : distance de projection (formule X=dx.x/z, Y=dy.y/z)
- `sx, sy` : demi-largeur et demi-hauteur de l'écran
- `zclip`, `zmiddle`, `zmax` : distances de clipping proche de début de brouillard et de clipping lointain.

`I M3setCamera(session S3d, objet H3d, parameters [[I I] [I I] [I I I]])`
Règle les paramètres d'une caméra : `[ [dx dy] [sx sy] [zclip zmiddle zmax] ]`.

`[H3d r1] M3ListOfBigFathers (session S3d)`
retourne la liste de tous les objets sans père, c'est-à-dire ceux qui sont au sommet d'une hiérarchie.

`[[I I I] [[I I I] [I I I] [I I I]]] M3calcPosRef (session S3d, objet H3d, référence H3d)`
calcule la position de l'objet dans le repère de l'objet référence. La matrice est constituée d'entiers 32bits, avec 16bits après la virgule.

`[I I I] M3angularFromMatrix ( matrix [[I I I] [I I I] [I I I]] )`
calcule un triplet d'angles correspondant à la matrice de passage. La matrice est constituée d'entiers 32bits, avec 16bits après la virgule.

`I M3getRadius (session S3d, objet H3d)`
lit le rayon de l'objet (distance maximale du centre du repère de l'objet à ses vertices)

`[x I y I z I r I] M3calcProj (session S3d, camera H3d, objet H3d)`
calcule la projection d'un objet sur une caméra et retourne, si l'objet est visible, les coordonnées x et y écran, la distance z et le rayon apparent r. Retourne `nil` si l'objet n'est pas visible ou en cas d'erreur. Ce calcul ne tient compte que du clipping z (front et back). Il ne tient pas compte des clipping latéraux.

[I I I] M3angularTarget (src_vector [I I I], dest_vector [I I I])
retourne les positions angulaires permettant à la source de pointer vers la destination. Le troisième angle est toujours nul.

I M3getObjType (session S3d, objet H3d)
Retourne le type de l'objet :

- M3_SHELL
- M3_CAM
- M3_MESH
- M3_COLL
- M3_LIGHT

[I I I] M3getGlobalVec (session S3d, repere H3d, vecteur [I I I])
Donne les coordonnées globales d'un vecteur exprimé dans le repère. Par coordonnées globales, on entend coordonnées dans le repère situé au sommet de l'arbre auquel appartient le repère passé en paramètre.

7.3.4. Gestion des matériaux

HMat3d M3getMat(session S3d, name S)
Retourne le handle d'un matériau, en fonction de son nom. `nil` si introuvable.

S M3materialName(session S3d, material HMat3d)
Retourne le nom du matériau.

I M3renameMat(session S3d, material HMat3d, name S)
Change le nom d'un matériau.

I M3fillMat(session S3d, material HMat3d)
Charge s'il y a lieu la texture d'un matériau.

I M3fillMatObj(session S3d, objet H3d)
Tente de charger toutes les textures utiles à un objet.

I M3recursFillMatObj(session S3d, objet H3d)
Tente de charger toutes les textures utiles à un objet et à sa descendance. Si l'objet est le sommet de la scène, tente de charger toutes les textures de la scène.

I M3getType(session S3d, material HMat3d)
Retourne le type par défaut d'un matériau (c'est-à-dire tel qu'il est souhaité par l'utilisateur).
Le résultat est une composante des masques suivants :

- MAT_TEXTURED : matériau texturé
- MAT_TRANSP : matériau avec effet de transparence
- MAT_LIGHT : matériau avec effet d'éclairage
- MAT_ENV : mapping environnemental
- MAT_GOURAUD : lissage gouraud

I M3getRealType(session S3d, material HMat3d)
Retourne le type courant d'un matériau (type par défaut, éventuellement altéré : par exemple, si la map n'est pas chargée, le type courant est forcé au flat). Voir ci-dessus pour les masques.

I M3setType(session S3d, material HMat3d, type I)

Définit le type par défaut d'un matériau en tenant compte des contraintes. Voir ci-dessus pour les masques.

I M3getMaterialFlat(session S3d, material HMat3d)
Retourne la couleur associée au matériau

I M3setMaterialFlat(session S3d, material HMat3d, color I)
Change la couleur associée au matériau

I M3getMaterialTransparency(session S3d, material HMat3d)
Retourne le coefficient de transparence associé au matériau (valeur entre 0 et 255)

I M3setMaterialTransparency(session S3d, material HMat3d, coef I)
Change le coefficient de transparence associé au matériau (valeur entre 0 et 255). N'est utilisé qu'un rendu flat.

I M3materialCount (session S3d, material HMat3d)
Retourne le compteur de références associé au matériau

[HMat3d r1] M3listOfMaterials(session S3d, objet H3d)
Retourne la liste des matériaux utilisés par l'objet.

HMat3d M3copyObjMaterial (session S3d, objet H3d, material HMat3d)
Crée un nouveau matériau, en remplacement et en copie d'un autre dans un certain objet. Retourne le nouveau matériau.

I M3chgObjMaterial (session S3d, objet H3d, current_material HMat3d, new_material HMat3d)
Remplace un matériau par un autre dans un objet donné.

7.3.5. Gestion des textures

HTx3d M3getTexture(session S3d, name S)
Retourne le handle d'une texture, en fonction de son nom. `nil` si introuvable.

S M3textureName(session S3d, texture HTx3d)
Retourne le nom de la texture.

I M3renameTexture(session S3d, texture HTx3d, name S)
Change le nom d'une texture.

HTx3d M3textureFromMaterial (session S3d, material HMat3d)
Retourne la texture associée au matériau

I M3textureCount (session S3d, texture HTx3d)
Retourne le compteur de références associé à la texture

I M3shiftTextureXY(session S3d, objet H3d, material HMat3d, x I, y I)
Effectue une translation sur les coordonnée de mapping des polygones de l'objet utilisant un certain matériau.

HTx3d M3copyMaterialTexture (session S3d, material HMat3d)
Crée une nouvelle texture, en remplacement et en copie d'une autre dans un certain matériau. Retourne la nouvelle texture.

I M3chgMaterialTexture (session S3d, material HMat3d, new_texture HTx3d)
Remplace la texture d'un matériau par une autre.

I M3isTextureFilled (session S3d, texture HTx3d)
Retourne 1 si le bitmap d'une texture est chargé, 0 sinon.

I M3fillTexture (session S3d, texture HTx3d)
Charge le bitmap d'une texture.

I M3freeTexture (session S3d, texture HTx3d)
Libère le bitmap d'une texture.

7.3.6. Gestion du rendu et lien avec l'interface 2D

H3d MX3render(session S3d, buffer ObjSurface, objet H3d, xplot I, yplot I, col I)
Calcule le rendu d'une scène à partir de la caméra de handler 'objet', dans un buffer de rendu. On spécifie la couleur de fond col (`nil` pour transparent). La fonction retourne le handler de l'objet visé par le point de coordonnées (`xplot,yplot`), `nil` si aucun. Attention, la taille de rendu de la caméra ne doit pas excéder la taille du bitmap buffer. La taille de rendu d'une caméra est le double des demi-hauteurs et demi-largeurs de la caméra. Utiliser `M3setCamera` pour modifier ces valeurs.

[H3d HMat3d] MX3renderm(session S3d, buffer ObjSurface, objet H3d, xplot I, yplot I, col I)
Calcule le rendu d'une scène à partir de la caméra de handler ic, dans un buffer de rendu. On spécifie la couleur de fond col (`nil` pour transparent). La fonction retourne le handler et le matériau de l'objet visé par le point de coordonnées (`xplot,yplot`), `nil` si aucun.

[H3d HMat3d I I I] MX3renderInfo(session S3d, objet H3d, xplot I, yplot I)
Retourne des informations sur le point (`xplot, yplot`) dans l'ordre suivant :

- handler 3d
- handler matériau
- coordonnée u de texture
- coordonnée v de texture
- n'effectue aucun rendu.

I M3blitTexture (session S3d, texture HTx3d, buffer ObjBitmap8)
Remplace le contenu d'une texture par un autre (en 8 bits), en écrasant la palette.

I M3blitTexture16 (session S3d, texture HTx3d, buffer ObjBitmap)
Remplace le contenu d'une texture par un autre (en 16 bits), en écrasant la palette.

ObjBitmap M3filter (buffer ObjBitmap, filter S)
applique un filtre sur un bitmap. Ce filtre est identique à celui utilisé lors du chargement de textures.
Exemple : filtre rouge 1/8 : Cff000020

7.4. Gestion des collisions

7.4.1. Principes

7.4.1.1. Eléments de collision

Le moteur 3D de la machine SCOL contient un système de gestion des collisions. Ce système repose sur des **éléments de collision**, que l'utilisateur peut définir de plusieurs manières : sphères, boîtes, polygônes. Les boîtes seront en fait des '**Obb'** (Oriented Bounding Boxes) : un Obb est un parallélépipède rectangle quelconque. Ses axes ne sont pas forcément ceux de la scène.

Ces éléments de collision n'ont pas de relation directe avec les éléments mesh de la scène. En effet, il est important de pouvoir d'une part définir des éléments de collision invisibles (par exemple des murs invisibles au bord d'une passerelle suspendue dans le vide, pour empêcher un utilisateur de tomber dans le vide), et d'autre part des éléments mesh qui n'entreront pas en compte pour le calcul des collisions, soit parce que l'utilisateur doit pouvoir passer au travers de ces éléments, soit par souci d'optimisation (soit l'utilisateur ne peut de toute manière pas se trouver à proximité de l'élément, soit on souhaite remplacer une géométrie compliquée par une simple boîte englobante).

Un système de détection d'intersection et de collision ne s'applique pas seulement à tester la position d'une caméra par rapport à un décor. Il s'applique en fait, de manière parfaitement générale, au test de deux ensembles d'éléments de collision.

A partir des éléments de collision, le moteur permet de détecter deux types d'événements :

- intersection : deux ensembles A et B d'éléments de collisions sont-ils en intersection ?
- collision en translation : étant donné un vecteur déplacement ***u***, et deux ensembles A et B d'éléments de collisions, y aura-t-il intersection entre A et B au cours du déplacement de A selon le vecteur ***u*** ? et si oui, à quel moment du déplacement et selon quel plan se produira la collision ?

7.4.1.2. Intégration des éléments de collision dans la scène

On distinguera élément de collision **primaire** et élément de collision **secondaire**.

Un élément de collision primaire est un élément fondamental de collision : ce sont les sphères, les boîtes, les polygones considérés comme des éléments solides de la scène, avec lesquels on effectue les tests d'intersection et de collision.

Certains éléments de collision primaires peuvent être regroupés dans des arbres binaires (chaque nœud a au plus deux fils). En fait les éléments de collision primaires sont situées dans les feuilles d'un tel arbre. Les nœuds qui ne sont pas des feuilles sont les éléments de collision secondaires. Ils sont de même nature (sphère, boîte, polygone), mais ne sont pas les éléments réels de collision : il englobent des éléments de collision primaire et permettent l'optimisation du calcul. L'algorithme (d'intersection ou de collision) part du principe que s'il y a intersection ou collision avec le père, alors il peut y avoir intersection ou collision avec les fils. Au contraire, s'il n'y a pas intersection ou collision avec le père, alors il ne peut y avoir intersection ou collision avec les fils.

Dans chaque nœud de l'arbre, on trouve non pas un élément de collision (primaire ou secondaire), mais une liste d'éléments de collision de même nature (des sphères, des boîtes ou des polygones). L'algorithme suppose de plus que si un fils est d'un type différent de son père (par exemple le fils est de type Obb, et le père de type Sphère), alors le fils est unique.

Les objets de collision de la scène sont en fait de tels arbres, éventuellement réduits à une seule feuille contenant un seul élément de collision. On peut donc définir autant d'objets de collisions que l'on souhaite dans une scène. Ces objets, comme les autres, sont définis par rapport au repère de leur

père, et peuvent eux-mêmes avoir des fils. Par exemple, on pourra définir un objet de collision de type Obb de la taille d'un objet (par exemple une table). L'objet de collision sera défini comme un fils de la table, ce qui permettra de déplacer simplement la table pour que l'objet de collision suive.

L'algorithme d'intersection ou de collision prend en entrée deux nœuds de la scène, notons-les A et B. Il effectue son calcul en considérant tous les objets de collision situés sous A et tous les objets de collisions situés sous B. Si A appartient à la descendance de B, le calcul se fera entre les objets situés sous A et les objets situés sous B à l'exception de ceux situés sous A. Typiquement, on pourra fournir à l'algorithme l'objet qui se déplace et l'ensemble de la scène.

7.4.1.3. Gestion des collisions

Une chose est de détecter une intersection ou une collision, une autre chose est de gérer un tel événement, autrement dit de proposer un déplacement, plus généralement une transformation, qui permet à la scène de quitter cet état d'intersection ou de collision. Ce problème est particulièrement complexe car il est spécifique de l'application et nécessite de choisir un modèle "physique" pour la scène. Ce n'est pas un problème purement géométrique. Le moteur 3d offre cependant une aide à la résolution des collisions. La fonction `M3testColl` de collision entre A et B retourne des informations complémentaires, liées au vecteur ***u*** :

- le déplacement géométriquement possible : un vecteur $\lambda.u$ avec λ entre 0 et 1.
- l'axe de collision ***v***, tel que ***u+v*** soit un vecteur qui transporte l'objet A à une certaine distance de garde de B, de manière à faire glisser A le long de B. Ce faisant, il n'est pas impossible que A entre en collision avec un autre objet C, ou même avec une autre partie de B.

7.4.2. API

`H3d M3createSphere(session S3d, rayon I)`
Crée un nœud de collision contenant une sphère dont le rayon est donné en paramètre.

`H3d M3createObb(session S3d, tailles [I I I])`
Crée un nœud de collision contenant une sphère et un Obb (l'Obb étant le fils de la sphère). Les tailles sont les demi-tailles de l'Obb.

`[[I I I] [I I I]] M3calcObb (session S3d, objet H3d)`
Calcule la position et les tailles de l'Obb recouvrant un objet.

`I M3firstRadius (session S3d, objet H3d)`
Si l'objet est un nœud de collision dont le sommet est une sphère, cette fonction retourne son rayon.

`[H3d H3d] M3testInter (session S3d, objetA H3d, objetB H3d)`
Test d'intersection entre les sous-arbres A et B. Si A (resp B) est sous-arbre de B (resp A), le test se fait entre le sous-arbre A (resp B) et le sous-arbre B (resp A) privé du sous-arbre A (resp B). Retourne `nil` si pas d'intersection, sinon retourne les deux objets en intersection.

`[H3d H3d [I I I] [I I I]] M3testColl (session S3d, objetA H3d, objetB H3d, vecteur [I I I], garde I)`
Test de collision entre les sous-arbres A et B. Si A (resp B) est sous-arbre de B (resp A), le test se fait entre le sous-arbre A (resp B) et le sous-arbre B (resp A) privé du sous-arbre A (resp B). Le test de collision se fait sur le déplacement de A selon un certain vecteur passé en paramètre, et exprimé dans le repère global. Retourne `nil` si pas de collision, sinon retourne les deux objets en collision ainsi que deux vecteurs :

- le premier est colinéaire au vecteur de déplacement et donne le déplacement jusqu'à la collision

- le second vaut :
  - `nil` si A et B sont en déjà en intersection avant le déplacement
  - sinon, un vecteur perpendiculaire au plan de collision, tel que, ajouté au vecteur de déplacement passé en paramètre, on obtienne un vecteur de proposition de résolution de la collision

8. PROGRAMMATION BIGNUM

8.1. Présentation générale

La librairie '*BigNum'* permet de manipuler de grands nombres entiers (allant jusqu'à 128 bits).

Elle offre diverses opérations telles que l'addition, la soustraction, la division euclidienne, la multiplication, l'exponentielle, etc...

La librairie offre des fonctions de conversion entre les chaînes de caractères et les '*BigNum'*, ce qui permet entre autres de sauvegarder ces nombres. La librairie permet également de convertir un texte complet vers une liste de '*BigNum'*, et inversement.

8.2. API

L'API définit un nouveau type : **BigN**

Fonctions de conversion :

`BigFromAsc [string S] BigN`
Cette fonction permet de créer un BigNum à partir d'une chaîne de caractère codée en hexadécimal.
Exemple :

- BigFromAsc "1" : définit le nombre 1
- BigFromAsc "f " **:** définit le nombre 15

`BigToAsc : fun [ nombre BigN ] S`
fonction inverse de la précédente

`BigToString : fun [ nombre BigN ] S`
convertit un bignum en une chaîne de caractère quelconque (la plus petite pouvant contenir le nombre). Le codage interne n'a pas à être précisé ici, cette fonction ne sert qu'à manipuler les bignums dans les communications ou dans les fichiers. La taille de la chaîne produite est environ deux fois plus petite que celle en Ascii de la fonction `BigToAsc`.

`BigToStringn : fun [ nombre BigN, taille I ] S`
idem fonction précédente, mais en spécifiant la taille de la chaîne devant être produite. Si celle-ci est trop courte, l'entier `BigNum` sera tronqué.

`BigFromString : fun [string S] BigN`
fonction inverse de `BigToString`

`BigRand : fun [ ] BigN`
choisit au hasard un nombre de 127 bits. Le hasard peut être manipulé par la fonction `srand` de la librairie standard.

`BigPrimal : fun [ i I ] BigN`
choisit au hasard un nombre premier de i bits de la forme 3n+2. Le hasard peut être manipulé par la fonction `srand` de la librairie standard.

`BigAdd : fun [x BigN, y BigN] BigN`

effectue l'addition : `x+y`.

`BigSub : fun [x BigN, y BigN] BigN`
effectue la soustraction : `x-y`.

`BigXor : fun [x BigN, y BigN] BigN`
effectue l'opération ou exclusif : `x^y`.

`BigMod : fun [x BigN, n BigN] BigN`
effectue l'opération : `x mod n`

`BigDiv : fun [x BigN, y BigN] BigN`
effectue la division euclidienne : `x div y`

`BigMul : fun [x BigN, y BigN] BigN`
effectue la multiplication (modulo 2^127) : `x*y`

`BigMuln : fun [x BigN, y BigN, n BigN] BigN`
effectue la multiplication : `x*y mod n`

`BigInvn : fun [x BigN, n BigN] BigN`
calcule l'inverse (théorème de Bezout) : `1/x mod n`

`BigExpn : fun [x BigN, y BigN, n BigN] BigN`
calcule l'exponentielle : $x^y \bmod n$

`BigPgcd : fun [x BigN, y BigN] BigN`
calcule le pgcd de x et de y.

`BigCmp : fun [x BigN, y BigN] I`
compare les entiers non signés x et y. retourne :

- 0 si x=y
- 1 si x>y
- -1 si x<y

`BigListFromString : fun [ text S, size_bloc I ] [ BigN r1 ]`
découpe un texte en une liste de mots de `size_bloc` octets chacun. Chaque mot est alors converti en un `BigNum`. La fonction complète aléatoirement la fin du message pour atteindre une taille multiple de `size_bloc`.

`BigListToString : fun [ liste [BigN r1], size_bloc I ] S`
fonction inverse de la précédente : reconstitue le texte original, à condition d'utiliser la même valeur de `size_bloc`.

8.3. Exemple

L'exemple suivant montre comment coder l'algorithme RSA. En supposant que l'exposant public vaut 3.

Les trois fonctions de l'API sont :

- `RSAcreate3` : calcule une paire de clef publique-clef privée
- `RSAcrypt3` : crypte un texte à l'aide de la clef publique

- `RSAdecrypt` : décrypte un texte à l'aide de la clef privée.

Attention, vérifiez que vous avez les autorisations légales avant de pouvoir intégrer cet exemple dans une de vos applications.

```
/*
    librairie RSA - juil 97 -
*/

/* fonctions a usage interne */
fun RSAsizebis(n,b,i)=
 if (BigCmp b n) >0 then i
 else RSAsizebis n BigAdd b b i+1;;

fun RSAsize(n)= RSAsizebis n BigFromAsc "1" 0;;

fun RSAcryptone3(l,n,i1)=
 if l==nil then nil else
 let l->[m nxt] in
   (BigToStringn (BigMuln (BigMuln m m n) m n) i1)::RSAcryptone3 nxt n i1;;

fun RSAcutstring(s,i,n)=
 let substr s i n -> z in
 if strlen z then z::RSAcutstring s i+n n
 else nil;;

fun RSAdecryptone(l,k,n)=
 if l==nil then nil else
 let l->[m nxt] in
  (BigExpn BigFromString m k n)::RSAdecryptone nxt k n;;

/* API */
/* decryptage d'un message s avec la cle privee n,k -> retourne une string
*/
fun RSAdecrypt(s,k,n)=
 let ((RSAsize n)-1)>>3 -> nbyte in
 BigListToString (RSAdecryptone (RSAcutstring s 0 nbyte+1) k n) nbyte;;

/* cryptage d'un message s avec la cle publique n -> retourne une string */
fun RSAcrypt3(s,n)=
 let ((RSAsize n)-1)>>3 -> nbyte in
 strcatn RSAcryptone3 (BigListFromString s nbyte) n nbyte+1;;

/* creation d'un couple clef publique-cle privee d'environ i bits (entre i
et i-1)
  retourne le couple [n k]*/
fun RSAcreate3(i)=
 let [BigFromAsc "1" BigFromAsc "3"] -> [un trois] in
 let [BigPrimal i>>1 BigPrimal i-(i>>1)] ->[p q] in
 if  ! BigCmp p q then RSAcreate3 i
 else let BigMul p q ->n in
 let BigMul BigSub p un BigSub q un -> phi in
 let BigInvn trois phi -> k in
  [n k];;
```

9. SQL

9.1. Présentation générale

La librairie SQL permet de manipuler n'importe quelle base de données, en utilisant le support '*ODBC*' et les requêtes SQL. Cette libraire n'est pas fournie dans le SCOL Voy@ger de base, car elle nécessite le support '*ODBC*', et celui-ci n'est pas présent sur toutes les machines. Cette librairie est fournie sous forme d'un plugin SCOL : **"SCOLsql.dll"**, et la ligne suivante doit être présente dans le fichier **usm.ini** :

```
plugin plugins/SCOLsql.dll SCOLloadSQL
```

Pour fonctionner, la base de données doit être définie et nommée dans le panneau de configuration de *Windows* ('*menu odbc32bits*'). Le nom que l'on donne alors au couple (fichier base de données/driver associé) sera alors le déterminant de la base, qui permettra à SCOL d'y accéder.

L'API comporte 3 fonctions et 2 types.

9.2. API

L'API définit un nouveau type : **SqlDB**

Ce type correspond à une connexion vers une base de données.

9.2.1. Fonction de création d'un connexion :

`SqlDB SqlCreate (Channel Chn, Nom_de_la_base S, Login S, Password S)`
Cette fonction retourne `nil` si la connexion échoue. Le nom de la base est celui défini dans le panneau de configuration, '*menu odbc32bits*'. Le '*login*' et le '*password*' sont les paramètres d'authentification à utiliser pour se connecter à la base. Si vous n'avez pas défini de droit d'accès à votre base, le login " admin " et le password " " devraient fonctionner.

9.2.2. Fonction de déconnexion :

`I SqlDestroy (connexion SqlDB)`
Cette fonction termine la connexion.

9.2.3. Fonction de requête :

`[[S r1]r1] SqlRequest (connexion SqlDB, requête_SQL S, paramètres [SqlParam r1])`
La requête SQL est donnée sous forme texte. Les paramètres de la requête y apparaissent sous la forme du caractère ' `?`'. La liste des paramètres (troisième argument de la fonction `SqlRequest`) respecte l'ordre et le nombre des ' `?`' dans la requête SQL.

Le résultat est une liste de listes de chaînes de caractères, qui correspond à la liste des réponses à la requête, où chaque réponse est une liste de colonnes.

Les paramètres sont passés sous forme d'une liste de paramètres de type **SqlParam**. Le type **SqlParam** est défini de la manière suivante :

```
typedef SqlParam =
 SQL_BIGINT S | SQL_BINARY S | SQL_BIT S | SQL_CHAR S |
```

```
SQL_DATE S | SQL_DECIMAL S | SQL_DOUBLE S | SQL_FLOAT S |
SQL_INTEGER S | SQL_LONGVARBINARY S | SQL_LONGVARCHAR S | SQL_NUMERIC S |
SQL_REAL S | SQL_SMALLINT S | SQL_TIME S | SQL_TIMESTAMP S |
SQL_TINYINT S | SQL_VARBINARY S | SQL_VARCHAR
```

9.3. Exemples

L'exemple suivant ouvre la base " *MyBase* " et effectue une recherche de '*passwor'd* en fonction d'un '*login'*" Foo ".

```
typeof db=SqlDB;;

fun main()=
set db=SqlCreate _channel "second" "admin" "";
SqlRequest db "SELECT Password FROM Table WHERE Table.Login=?;"
  (SQL_CHAR "Foo")::nil;
SqlDestroy db;
0;;
```

L'exemple suivant ouvre la base " *MyBase* " et effectue une recherche de '*login'* en fonction d'un numéro de téléphone " 1234 ".

```
typeof db=SqlDB;;

fun main()=
set db=SqlCreate _channel "second" "admin" "";
SqlRequest db "SELECT Login FROM Table WHERE Table.Tel=?;"
(SQL_NUMERIC "123")::nil;
SqlDestroy db;
0;;
```

L'exemple suivant ouvre la base " *MyBase* " et sort sur la console (`_fooS strbuild`) la liste des '*logins'* et des numéros de téléphone.

```
typeof db=SqlDB;;

fun main()=
set db=SqlCreate _channel "second" "admin" "";
_fooS strbuild SqlRequest db "SELECT Login,Tel FROM Table;" nil;
SqlDestroy db;
0;;
```

L'exemple suivant ajoute une nouvelle ligne dans la base.

```
typeof db=SqlDB;;

fun main()=
set db=SqlCreate _channel "second" "admin" "";
SqlRequest db "INSERT INTO Table VALUES(?,?,?);"
  (SQL_CHAR "Titi")::(SQL_CHAR "xyz")::(SQL_NUMERIC "789")::nil;
```

```
SqlDestroy db;
0;;
```

L'exemple suivant modifie le mot de passe de Titi.

```
typeof db=SqlDB;;

fun main()=
set db=SqlCreate _channel "second" "admin" "";
SqlRequest db "UPDATE Table SET Password=? WHERE Login=?;"
  (SQL_CHAR "123abc")::(SQL_CHAR "Titi")::nil;
SqlDestroy db;
0;;
```

L'exemple suivant suppime la fiche de Titi.

```
typeof db=SqlDB;;

fun main()=
set db=SqlCreate _channel "second" "admin" "";
SqlRequest db "DELETE FROM Logins WHERE Login=?;"
  (SQL_CHAR "Titi")::nil;
SqlDestroy db;
0;;
```

10. INTERFAÇAGE HTTP

La machine SCOL intègre le protocole http tant du côté serveur que du côté client. Cela signifie qu'il est possible en quelques lignes de SCOL de créer un petit serveur http, mais aussi d'effectuer en tant que client des requêtes http vers n'importe quel serveur http de la planète. Il est de même possible que deux machines SCOL communiquent via http, l'une faisant office de serveur, l'autre de client. Cela est très utile pour pouvoir se connecter sur un site SCOL lorsqu'on est soi-même derrière un firewall qui ne laisse passer que le protocole http.

Nous détaillons les deux aspects **serveur** et **client**.

10.1. Http serveur

10.1.1. Principes

On définit deux types :

- **HTTPserver** : serveur Http
- **HTTPcon** : connexion Http. Lorsqu'une requête est reçue par un serveur http, celui-ci crée une "connexion" qui permettra au programme serveur de répondre à la requête.

Pour définir un serveur http, il suffit de :

- choisir un port Tcp-Ip libre
- définir une callback, fonction qui sera appelée chaque fois qu'une requête sera reçue par le serveur.

La fonction à appeler est :

```
startHTTPserver : fun [Chn I fun [HTTPcon u0 S] S u0] HTTPserver
```

Par exemple :

```
startHTTPserver _channel 8080 @callback nil
```

La fonction de callback est appelée lorsque le serveur a reçu la requête complète. Elle prend trois arguments :

- la connexion http
- le paramètre utilisateur
- une chaîne de caractère contenant la requête

Elle doit retourner une chaîne de caractère qui est la réponse à transmettre. C'est le mode synchrone : la callback retourne immédiatement la réponse à transmettre.

Il existe un mode asynchrone, qui permet au serveur de reporter sa réponse. Pour cela, il suffit que la callback retourne `'nil'`. Le programme serveur peut ensuite :

- envoyer la réponse, paquet par paquet, en utilisant la fonction `HTTPsend`
- puis fermer la connexion après le dernier paquet, en utilisant la fonction `closeHTTPcon`
- il peut également envoyer le contenu d'un fichier, qui sera transmis directement du disque vers la connexion Http, sans passer par la mémoire de la machine SCOL. A la fin du transfert, la connexion Http sera fermée automatiquement. Ceci se fait avec la fonction `HTTPsendFile`.

10.1.2. Quelques conseils

Une requête Http contient :

- un en-tête dont la première ligne est de la forme VERBE URL, le verbe étant généralement `GET` ou `POST`
- une fin d'en-tête : `"\13\10\13\10"` (deux retours à la ligne)
- éventuellement un corps de message, utilisé pour les requêtes `POST`

Pour parser une telle requête, le plus simple est d'utiliser un `strfind` pour trouver la fin de l'en-tête, et d'appliquer la fonction `strextr` sur l'en-tête.

La réponse à une requête Http doit également débuter par un en-tête. Par exemple, on pourra prendre l'en-tête suivant pour un message texte :

```
HTTP/1.0 200 OK\13\10Server: SCOL HTTP server\13\10Content-Type:
text/html\13\10\13\10
```

On oublie souvent d'envoyer cet en-tête, donc attention à ne pas perdre de temps sur cette faute d'étourderie.

Exemple :

```
var http_header="HTTP/1.0 200 OK\13\10Server: SCOL HTTP
server\13\10Content-Type: text/html\13\10\13\10";;

fun http_onrequest(con,db,req)=
  let hd strextr req -> l in
  let l->[com [url _]] in
  if (!strcmpi com "GET") then strcat http_header "GET"
  else if (!strcmpi com "POST") then strcat http_header "POST"
  else "";;

...
 startHTTPserver _channel 8080 @http_onrequest nil;
...
```

Dans cet exemple, on retourne une page qui dépend du verbe de la requête. On remarquera comment le parsing a été effectué. Si le verbe n'est ni `GET` ni `POST`, on retourne une chaîne vide. Le client recevra donc cette chaîne vide, sans header, ce qui provoquera une erreur, ce qui peut être voulu comme ici.

10.2. Http client

10.2.1. Principes

Pour effectuer une requête http, le client doit spécifier le verbe, l'url et éventuellement un corps de message.

Dans le cas d'un `GET`, seule l'url doit être précisée, d'où une Api simplifiée dans ce cas. On utilisera la fonction suivante :

```
INETGetURL : fun [Chn S I fun [INET u0 S I] u1 u0] INET
```

Les arguments sont :

- le canal propriétaire
- l'url
- un flag (le laisser à 0)

- une callback
- un paramètre utilisateur.

La fonction retourne un type **INET** qui correspond à une requête en cours, ce qui permettra de l'interrompre si l'on ne souhaite plus recevoir la réponse.

La solution la plus simple pour la majorité des cas consistait à n'appeler la callback qu'une fois la réponse complètement reçue. Cependant sur internet, on trouve certaines applications (notamment du pseudo-streaming) qui retournent la réponse en plusieurs morceaux : il peut s'écouler plusieurs minutes entre le début et la fin de la réponse. Pour cela, il a été jugé préférable d'informer le programme SCOL chaque fois que des données ont été réceptionnées. La callback prend donc les 4 arguments suivants :

- requête `INET`
- paramètre utilisateur
- données reçues
- état de la requête :
  - 0 : des données viennent d'être reçues, elles sont dans le troisième argument
  - 1 : c'est la fin de la requête (le troisième argument vaut `nil`)
  - 2 : erreur réseau (le troisième argument vaut également `nil`)

Exemple :

```
fun cbgethttp(inet,z,s,reason)=
 let z->[content] in
  if reason==0 then (mutate z <- [strcat content s];nil)
  else if reason==1 then (_fooS content; _fooS "OK");
  else _fooS "ERROR";;

...
INETGetURL _channel "http://www.cryo-networks.com" 0 @cbgethttp [nil];
...
```

Il est difficile de détecter les erreurs réseau. En effet, lorsqu'un client http est derrière un proxy et demande une url qui n'existe pas, le client demande en fait cette url au proxy qui se charge de la demander au serveur internet correspondant à l'url. Si la page n'existe pas, c'est le proxy qui en sera informé. Celui-ci crée alors souvent une page html parfaitement valide contenant un petit texte qui explique qu'une erreur est survenue. Mais pour le client http, il s'agit d'une réponse en règle, et non d'une erreur.

Si vous développez une application dans laquelle vous maîtrisez le client http et le serveur http, il est recommandé que le client puisse déterminer facilement si la page qu'il a reçue était bien celle qu'il attendait, par exemple en la faisant précéder d'un *'magic number'*.

10.2.2. Méthode POST

Nous venons de voir comment réaliser une requête `GET`, passons maintenant à la suite, et en particulier à la méthode POST. Il s'agit ici de pouvoir passer un corps de message, ce qui se fait en utilisant la fonction :

```
INETGetURLex : fun [Chn S S S I fun [INET u0 S I] u1 u0] INET
```

Les arguments sont :

- le canal propriétaire
- le verbe (POST)
- l'url
- le corps de message

- un flag (le laisser à 0)
- une callback
- un paramètre utilisateur.

La réception de la réponse – et le fonctionnement de la callback—est identique à la fonction `INETGetURL`.

Il est possible d'interrompre une requête en cours en appelant la fonction :

```
INETStopURL : fun [INET] I
```

11. PROGRAMMATION MULTIMEDIA

SCOL offre de nombreuses possibilités multimédia en plus de la 3d et des interfaces 2d classiques.

Il faut noter toutefois que certaines de ces possibilités sont parfois très dépendantes de la plateforme. Ainsi certaines api ne sont pas implémentées sur les plateformes Unix : les fonctions sont alors présentes sous leur forme vide (même type, mais la fonction retourne toujours `nil`).

On se reportera au manuel de référence pour le détail de chaque Api multimedia.

11.1. Api RealPlayer

Cette Api permet de lire un flux RealPlayer (audio et/ou vidéo). Le programme définit simplement l'url du document à lire, il est ensuite notifié de l'état d'avancement. Le son est traité automatiquement et transmis au haut-parleur. Les images sont passées au programme dans une bitmap, et le programme en fait ensuite ce qu'il veut : affichage 2d, utilisation de la bitmap comme texture pour appliquer la vidéo sur un objet 3d, …

L'Api offre également des outils pour contrôler le flux (lecture, pause, positionnement, volume, …).

En outre, l'Api permet d'exploiter le système de login/password défini par Real.

11.2. Api Quicktime

Cette Api applique le même principe que pour RealPlayer. Les fonctions diffèrent cependant car les deux technologies se présentent de manière fondamentalement différente pour le développeur.

11.3. Api multimédia basique

Il s'agit d'utiliser les players multimédia présents sur le système pour jouer des fichiers multimédia classiques comme Wav, Avi, Mpeg, …

En fait, la machine SCOL transmet les fichiers au système d'exploitation. Celui les exploitera s'il dispose des bons drivers.

11.4. Api audio

L'Api audio offre plusieurs composantes :

- enregistrement/lecture mono
- compression/décompression d'échantillon
- lecture mixée de son éventuellement 3d

11.4.1. Enregistrement/lecture mono

Cette Api permet d'enregistrer du son en précisant la fréquence d'échantillonnage et la taille des échantillons. Elle permet également de lire ces échantillons, en précisant de nouveau la fréquence et la taille.

Cette Api tire parti des possibilités full-duplex si votre matériel en est doté. Dans le cas contraire, les fonctions de démarrage de lecture ou d'enregistrement retourneront des erreurs si vous essayez de lancer les deux simultanément.

11.4.2. Compression/décompression audio

Cette Api permet de compresser et de décompresser des échantillons audio. La méthode de compression est adaptée à de très faibles débits, puisqu'elle permet de descendre à un codage de 8Kb/s, ce qui représente un sixième de la bande passante d'un modem 57,6Kb/s

11.4.3. Lecture mixée de son 3d

Cette Api (utilisant '*DirectSound*' sous *Windows*), permet de lire simultanément plusieurs échantillons, tout en pouvant spécifier pour chacun une position 3d.

11.5. Api vidéo

La machine SCOL est capable via cette Api d'utiliser une caméra vidéo, et de réaliser l'acquisition d'images, en précisant la fréquence souhaitée d'acquisition.

L'utilisateur définit une fonction callback qui recevra régulièrement les images. Il est alors possible :

- de les convertir en bitmap, pour faire de l'affichage, ou bien sauver ces images
- de les compresser pour éventuellement les transmettre.

La librairie offre à cet effet des fonctions de compression/décompression vidéo très intéressantes.

11.6. Impression

A la limite du multimédia, on trouve les fonctionnalités d'impression. SCOL permet deux types d'impression : l'impression de texte brut, ou l'impression de bitmaps.

Dans le cas de l'impression de texte brut, on passera simplement les chaînes de caractère à imprimer.

Dans le cas de l'impression de bitmap, on passera la bitmap ainsi que les coordonnées et la taille d'impression (avec une précision au micron).

12. MACHINE SCOL : LANCEMENT, CONTROLE, CLIENT ET SERVEUR STANDARD

Nous avons jusqu'à présent présenté le fonctionnement interne de la machine SCOL. Nous allons présenter une vision plus globale, concernant le lancement et le contrôle d'une machine SCOL. De plus, une manière standard de connexion a été mise au point pour définir la notion de browser, nous la détaillerons dans cette partie.

12.1. SCOL Voy@ger : le superviseur

Lorsque vous lancez un de vos programmes SCOL, il y a toujours une autre machine SCOL qui se lance et qui se caractérise par l'affichage d'une fenêtre appelée SCOL Voy@ger. Le SCOL Voy@ger permet à l'utilisateur de réaliser certaines opérations, mais il joue avant tout le rôle de **superviseur** : il permet de gérer les différentes machines SCOL présentes sur la machine, à travers une interface utilisateur graphique.

Le superviseur se met en route automatiquement au lancement de la première machine SCOL. Chaque machine SCOL est reliée au superviseur via une socket Tcp-Ip locale, appelée **life-socket** : la fermeture de cette socket par le superviseur entraîne l'arrêt de la machine SCOL. Inversement, la fermeture de cette socket par la machine SCOL informe le superviseur de l'arrêt de la machine SCOL. Cette socket permet en outre au superviseur de communiquer avec les machines SCOL locales, par exemple pour commander l'ouverture ou la fermeture de la fenêtre console.

Le superviseur est une machine SCOL normale. Sa particularité réside dans le fait que les autres machines SCOL essaient automatiquement de tendre un canal vers elle.

L'utilisation exacte de la machine SCOL est la suivante :

- **a. lancée sans argument** : Si le superviseur est déjà présent, rien ne se passe. Sinon la machine SCOL se lance en mode superviseur et assure alors cette fonction : voir plus loin les détails.
- **b. lancée avec un argument** : Si le superviseur est absent, une autre machine SCOL est lancée, qui se met en mode superviseur

Puis la machine SCOL est lancée avec l'argument qui donne le script de démarrage et éventuellement les droits et la taille mémoire : voir plus loin les détails.

12.2. Lancement du superviseur

Le superviseur présente une interface qui offre les fonctionnalités suivantes :

- lancement de machines SCOL dont le script de démarrage est choisi par l'utilisateur dans un menu déroulant.
- maintient d'un certain nombre de machines SCOL : celles-ci sont lancées automatiquement au démarrage du superviseur, et sont relancées lorsqu'elles tombent.
- visualisation des machines SCOL en fonctionnement, avec nom du script de démarrage et temps écoulé depuis le lancement.
- possibilité de détruire une machine SCOL particulière et d'afficher/cacher la fenêtre console.
- destruction de toutes les machines SCOL lors de la fermeture du superviseur

Les fichiers devant être présents dans le répertoire SCOL (typiquement '*C:/Program Files/SCOL*') sont :

- **usmress.ini** : fichier texte contenant la définition initiale des variables ressources
- **usm.ini** : fichier texte, contenant des paramètres " durs " :
  - `echo` : masque d'affichage
  - `port` : port utilisé par le superviseur (1200 par défaut)
  - `log` : activation des fichiers logs
  - `logwin` : affichage de la console en cas d'erreur runtime. Si réglé sur '`no`', la machine s'arrête immédiatement lors d'une erreur (ceci est utile lorsqu'on travaille sur une machine SCOL distante)
  - `forcedIP` : force la définition de l'adresse IP locale (ceci est utile lorsqu'un serveur ne connaît pas son adresse IP apparente depuis internet)
  - `HTTPproxy` : proxy http
  - `update` : date de la version de SCOL
  - `SCOL` : nom de la dll SCOL en cours d'usage
  - `plugin` : nom d'un plugin pour SCOL (cette ligne peut apparaître plusieurs fois)
  - `disk` : partition SCOL (cette ligne peut apparaître plusieurs fois)

12.3. Lancement d'une machine SCOL avec script de démarrage

Dans cette partie, on décrit la ligne de commande de lancement d'une machine SCOL.

La ligne de commande de la machine SCOL peut contenir jusqu'à trois arguments (seul le premier est indispensable):

- le premier donne le script de démarrage,
- le deuxième donne les droits de la machine (voir plus bas),
- la taille mémoire à allouer à la machine virtuelle (obsolète).

12.3.1. Script de démarrage

La machine SCOL utilise un fichier script de démarrage qui peut être soit un fichier, soit une chaîne de caractères. La syntaxe des scripts a été définie dans le chapitre Canaux et communications.

Au démarrage, la machine crée un premier canal **unplugged**, avec un l'**environnement minimal**. Le script est alors exécuté dans ce canal.

Le script de démarrage est a priori très court : chargement d'un ou plusieurs packages SCOL, puis exécution d'une commande.

Le script de démarrage peut être exprimé de deux manières :

12.3.1.1. Fichier Script

Dans ce mode on passe simplement le nom du fichier script, dont le suffixe est '**.SCOL*'. Par exemple :

```
C:\Program Files\SCOL\usmwin.exe C:\Program Files\SCOL\Partition\test.SCOL
```

Le nom d'une telle machine sera '`test.SCOL`'

12.3.1.2. Chaîne Script

Dans ce mode, on passe directement le contenu du script, exprimé d'une manière un peu particulière :

- les caractères alphanumériques sont conservés
- les espaces sont remplacés par des '`+`'
- les autres caractères sont remplacés par un '`%`' suivi de 2 chiffres hexadécimaux

Ce "script" est précédé du nom de la machine entouré de caractères `$`.

Par exemple :
```
C:\Program Files\SCOL\usmwin.exe
$Tutorial/mytest.SCOL$%5fload+%22locked%2flib%2fconst%2epkg%22%0d%0a%5fload
+%22Tutorial%2fmytest%2epkg%22%0d%0amain%0d%0a
```

12.3.2. Droits d'exécution

Le deuxième argument de la ligne de commande donne les droits de la machine (ceux-ci sont rarement utilisés). C'est une chaîne de caractère telle que : `CSDMRWK`

La machine SCOL dispose de certains droits, chacun représentés par une lettre :

- `C` : client-networking (accès à la fonction `_openchannel` avec une adresse différente de `nil`)
- `S` : server-networking (accès à la fonction `_setserver`)
- `D` : distant-networking (accès à la fonction `_openchannel` avec une adresse distante)
- `M` : opening new machine without cache (lancement d'une machine sans que le cache ne soit activé)
- `R` : normal file reading (accès en lecture aux fichiers non signés)
- `W` : normal file writing (accès en écriture aux fichiers non signés)
- `K` : signed file writing (accès en écriture aux fichiers signés)

Les constantes `C_Rights`, `S_Rights`, ..., `K_Rights` sont définies dans le langage SCOL (ce sont des masques entiers). Une machine SCOL peut connaître ses paramètres :

`_getrights : fun [ ] I`
retourne les droits de la machine (à exploiter avec les masques précédents)

`_setrights : fun [ I ] I`
définit de nouveaux droits dans le sens de la restriction (nécessairement inférieurs à ceux de la machine)

12.3.3. Mémoire

Le troisième et dernier argument de la ligne de commande donne la taille de la mémoire utilisée par la machine SCOL. Ce paramètre est désormais obsolète, depuis que la machine virtuelle gère dynamiquement la taille de sa mémoire. La machine virtuelle commence désormais toujours avec 1Mo de mémoire et augmente cette taille au fur et à mesure des besoins.

`_sizememory : fun [ ] I`
retourne la taille de la mémoire de la machine SCOL (comptée en mots de 32 bits).

`_freememory : fun [ ] I`
retourne la taille de la mémoire libre de la machine SCOL (comptée en mots de 32 bits). Cette fonction déclenche un Garbage Collector et est donc relativement lente.

12.4. Lancement d'une machine ou d'un autre processus par une machine SCOL

Une machine SCOL peut en lancer une autre grâce à la commande suivante :

`_newmachine : fun [ S1 S2 I1 I2 ] I`

Crée une nouvelle machine dont le nom est S_1, dont le script de démarrage est S_2, dont les droits sont I_1 et la taille mémoire est I_2. Evidemment les droits de la nouvelle machine sont au plus égaux à ceux de l'ancienne.
Si I_1 ou I_2 valent **`nil`**, la nouvelle machine hérite des valeurs de l'ancienne.

`_newmachineS : fun [ P I1 I2 ] I`
Crée une nouvelle machine dont le fichier script est P, dont les droits sont I_1 et la taille mémoire est I_2. Evidemment les droits de la nouvelle machine sont au plus égaux à ceux de l'ancienne.
Si I_1 ou I_2 valent **`nil`**, la nouvelle machine hérite des valeurs de l'ancienne.

`_openbrowserhttp : fun [S] S`
historiquement cette fonction commande l'ouverture du browser standard de votre machine, en passant en argument une URL. En fait, il est possible de faire plus avec cette fonction, tout dépend du préfixe de votre URL :

- `http://` : lance le browser standard de votre machine
- `ftp://` : lance le browser standard de votre machine
- `mailto:` : lance le browser standard de votre machine
- `file://` : lance le fichier après que l'utilisateur a donné son accord au travers d'une boîte de dialogue.

Exemples : pour utiliser la fonction `_openbrowserhttp`, inutile d'écrire un programme, utilisez le SCOL Voy@ger et entrez comme URL les exemples suivants :

```
http://www.cryo-networks.com
```

```
mailto:SCOL.info@cryo-interactive.com
```

```
file://C:/Windows/calc.exe
```

12.5. Communication entre la machine SCOL et le superviseur

La machine SCOL est reliée au superviseur par un canal normal. Ce canal est défini comme étant **_masterchannel**, et la socket correspondante est appelée **socklife**.

La variable SCOL **_masterchannel** est accessible à l'utilisateur. Il peut s'en servir pour envoyer des messages au superviseur.

Exemple :
```
defcom Copen=open S;;

...
_on _masterchannel Copen ["http://www.cryo-networks.com"];
...
```

Le message '`open`' demande au superviseur de lancer une connexion vers une url (http ou SCOL).

Le message '`goto`' fait la même chose, puis ferme la machine SCOL émettrice.

La socket `socklife` présente une particularité : lorsqu'elle tombe (fermeture ou erreur), la machine SCOL s'arrête.

12.6. Serveur et Client Standard

12.6.1. Généralités sur la communication des machines SCOL

Les machines SCOL communiquent en échangeant des messages de la forme commande+arguments (voir la partie sur les canaux et les communications). Lorsqu'un message atteint sa cible, celle-ci recherche si une fonction porte le même nom que la commande, et si oui, exécute cette commande.

En fait, lorsqu'une machine SCOL envoie un message à une autre, rien ne l'assure a priori que son correspondant a défini une fonction qui porte le nom de la commande contenue dans son message. On peut définir une notion de '**vocabulaire**' : le vocabulaire de sortie d'une machine SCOL correspond à l'ensemble des commandes que celle-ci est susceptible de produire. Le vocabulaire d'entrée d'une machine SCOL correspond à l'ensemble des commandes que celle-ci est capable d'interpréter, à savoir l'ensemble des fonctions commençant par un double-underscore.

Pour que deux machines puissent communiquer entre elles, il faut que le vocabulaire de sortie de l'une corresponde au vocabulaire d'entrée de l'autre et inversement. Sinon, les messages envoyés par l'une ne seront pas interprétés par l'autre : si Alice lance un message '`f 1`' à Bob, il faut que Bob possède la fonction `__f` sur sa machine (et plus précisément dans l'environnement du canal que le relie à Alice).

La particularité de SCOL est que le vocabulaire d'une machine est dynamique : il est possible à tout moment de modifier l'environnement d'un canal, soit en ajoutant des packages, soit en retranchant des packages. Cela signifie bien qu'il est possible de modifier à tout moment le vocabulaire d'entrée et de sortie de la machine.

Lorsqu' Alice appelle la machine de Bob, Alice ne sait pas a priori quel vocabulaire Bob utilise, mais dès que Bob lui aura dit quels fichiers compiler sur son canal, Alice pourra modifier son vocabulaire et le rendre compatible avec celui de Bob. Malheureusement, Bob ne peut dire à Alice quels fichiers compiler, puisque pour cela il a déjà besoin d'un vocabulaire commun.

Pour résoudre ce problème, on a défini ce qu'on appelle le serveur et le client standards, qui sont en fait des fichiers écrits en langage SCOL. Leur rôle est d'initier une communication entre deux machine SCOL. Lorsque Alice appelle Bob, les opérations suivantes se déroulent :

- Alice utilise le client standard sur le canal de connexion vers Bob
- lorsque Bob reçoit la demande de connexion d'Alice, il place sur ce canal le serveur standard
- le serveur standard donne alors au client la liste des packages nécessaires pour aller plus loin
- le client standard vérifie qu'il possède ces packages
- si certains lui manquent, il en informe le serveur qui les lui transmet
- une fois les packages nécessaires en possession du client, le serveur lui transmet un script de démarrage
- une fois le script transmis, le serveur retire le package serveur standard et exécute un script, appelé script serveur
- de même, le client retire les packages client standard et exécute le script que lui a transmis le serveur, appelé script client.
- à ce moment, les vocabulaires d'Alice et de Bob sont bien synchronisés (si toutefois Bob a transmis les bons packets). L'application réelle du serveur de Bob peut alors commencer.

12.6.2. Serveur Standard - version 3

Comme on vient de le voir, le serveur qui utilise le serveur standard doit fournir plusieurs éléments :

- il doit placer initialement le serveur standard version 2 sur les canaux qui le relient à ses clients. Cela se fait dans la déclaration `_setserver`. Exemple pour un serveur sur le port 1285 :
  ```
  _setserver _envchannel _channel 1285 "_load \"locked/stdsrv3.pkg\"";
  ```
- il doit fournir un script serveur. Il suffit de déclarer la variable `scriptserver` :
  ```
  var scriptserver="_load \"fs/fssrv2.pkg\"\n_contact";;
  ```
- il doit fournir un script client. Il suffit de déclarer la variable `scriptuser` :
  ```
  var scriptuser="_load \"fs/fscli.pkg\"\nmain
  ```
- il doit fournir la liste des packages nécessaires au client. Il suffit de déclarer la variable `packsusers` qui est une liste de type `[[S S S] r1]` : la première chaîne est le nom d'un package, les deux autres doivent être des chaînes vides.
  ```
  var packsusers=["fs/fscli.pkg" "" ""]::nil;;
  ```
- il doit fournir le numéro de version minimale du SCOL Voy@ger des clients supporté par le site.

  `var versionuser=0;;` Ce numéro de version sera comparé avec celui retourné par la fonction `_version`.

Exemple :

Le client a besoin du fichier "*sample/foo.pkg*". Le script du client est `_load "sample/foo.pkg"\nstart`. Le script du serveur est `_load "sample/bar.pkg"\nstart`. Alors le serveur s'écrit :

```
/* server */
var packsusers=["sample/foo.pkg" "" ""]::nil;;
var scriptserver="_load \"sample/bar.pkg\"\n_contact";;
var scriptuser="_load \"sample/foo.pkg\"\n_contact";;
var versionuser=0;;

fun main()=
_setserver _envchannel _channel 1285 "_load \"locked/stdsrv2.pkg\"";;
```

Pour ceux que cela intéresse :

- le serveur standard utilise le fichier : `locked/stdsvr3.pkg`
- le client standard utilise les fichiers : `locked/stduser.pkg`, `locked/stduser1.pkg`, `locked/stduser2.pkg` et `locked/IPrequest.pkg`

Nous verrons dans la partie 'intégration dans une page web' comment lancer " manuellement " le client standard, mais en fait le SCOL Voy@ger s'en charge à votre place lorsque vous entrez une Url sans préfixe, ou commençant par `SCOL://`

Note : il existe des versions 1 et 2 du serveur standard, dont le nom du fichier est "`locked/stdsvr.pkg`" et "`locked/stdsvr2.pkg`". La version 3 est similaire à la version 2, mais utilise la nouvelle compression '`zip`' au lieu de '`mzip`'.

13. POSSIBILITES D'INTEGRATION

SCOL offre de nombreuses passerelles vers d'autres technologies :

13.1. Interfaçage par échange de fichiers

C'est évidemment la méthode la plus évidente, qui s'utilise principalement côté serveur : la machine SCOL ayant accès à tous les fichiers présents dans les partitions SCOL, il suffit que la technologie que vous voulez interfacer avec SCOL produise ou exploite des fichiers dans ces partitions, et que votre serveur SCOL produise ou exploite les même fichiers.

13.2. Interfaçage par base de données : librairie SQL

Ceci s'utilise uniquement côté serveur : le serveur accède aux données d'un autre système d'information, au travers de bases de données SQL (via la technologie Odbc)

13.3. Interfaçage par http : librairies http serveur ou client

Côté serveur SCOL, on utilise indifféremment les deux librairies :

- **http client** : la technologie que vous voulez interfacer avec SCOL est une technologie web (cgi, asp, …). Alors votre serveur SCOL peut lui-même devenir client de cette technologie web en effectuant lui-même les requêtes http grâce à la librairie http client.
- **http serveur** : la technologie que vous voulez interfacer avec SCOL est une technologie cliente web (on le trouve souvent dans les systèmes de paiement par exemple : le serveur de paiement fait lui-même, ou via le browser web de l'internaute, une requête http indiquant le résultat de l'opération : réussite ou échec). Votre serveur SCOL peut alors recevoir et traiter ces requêtes en ouvrant un serveur http grâce à la librairie http serveur.

Côté client SCOL, on utilisera principalement la librairie http client. Ceci permet au client SCOL d'accéder à des services web quelconques : effectuer une requête et analyser la réponse. Par exemple pour représenter en 3d le résultat d'une recherche sur un moteur comme Altavista ou Yahoo, le client SCOL fera la requête de recherche, recevra la réponse, l'analysera, et la convertira en une représentation 3d.

13.4. Intégration dans une page web

13.4.1. Interfaçage simple

Il est possible de faire fonctionner SCOL en tant que plug-in pour Netscape ou en tant que composant '*ActiveX*' pour MSIE ou pour '*visual Basic*'.

Pour intégrer SCOL dans une page Web, il suffit de mettre le code suivant (l'exemple ici décrit une zone de 500 points par 400) :

```
<body onLoad="Load()" bgcolor="#FFFFFF" text="#000000" link="#00CCCC">
<object
id="SCOL"
name="SCOL"
classid="clsid:7A96FF35-4937-11D1-8F2C-00609779BDA3"
codebase="http://www.cryo-networks.com/files/atlSCOL.dll"
```

```
align="middle"
border="0"
width="500"
height="400">
<embed
align="baseline"
border="0"
width="500"
height="400"
name="SCOL"
pluginurl="http://www.cryo-networks.com/files/scp10.exe"
pluginspage="http://www.cryo-networks.com/files/scp10.exe"
type="application/x-SCOL"></embed>
</object>

<script LANGUAGE="JavaScript">
<!--
function Load()
{
  if(navigator.appName=='Netscape')
  {
document.SCOL.LaunchMachine('$browser$%5fload+%22locked%2fstduser%2epkg%22%
0amain+%22SCOL%2ecryopolis%2ecom%3aCryopolis%22+ffffffff+NIL CSDMRWK
262144',1,0);
  }
  else
  {
SCOL.LaunchMachine('$browser$%5fload+%22locked%2fstduser%2epkg%22%0amain+%2
2SCOL%2ecryopolis%2ecom%3aCryopolis%22+ffffffff+NIL CSDMRWK 262144',1,0);
  }
}
//-->
</script>
```

En effet, pour lancer la machine SCOL, les choses diffèrent entre Netscape et MSIE, mais dans les deux cas vous utilisez une fonction `LaunchMachine`, en passant en argument la ligne de commande, ici un client "standard user" vers le port 220.87.26.15:3005

On remarquera la syntaxe de l'argument des fonctions `Launchmachine`. Il s'agit d'une chaîne de caractères comptant trois mots séparés par des espaces :

- script de lancement de la machine : de type `'$nom$script'`, le script étant au format " `strtoweb` ".
- droits de la machine (généralement CSDMRWK)
- taille de la mémoire donnée à la machine.

Il y aura généralement deux types de scripts :

- le script lançant la machine SCOL dans une page web, il s'agit alors du script " stduser " :

  ```
  _load "locked/stduser.pkg"
  main "url" ffffffff NIL
  ```

  Ce qui en appliquant la fonction `strtoweb` (remplace les espaces par des `+` et tout caractère non alpha-numérique par '*%code_ascii_en_hexadécimal*'), pour l'url `SCOL.cryopolis.com:Cryopolis` donne :
  `%5fload+%22locked%2fstduser%2epkg%22%0amain+%22SCOL%2ecryopolis%2ecom%3aCryopolis%22+ffffffff+NIL`

Cependant, la machine SCOL appliquant la fonction `webtostr` sur cette chaîne, fonction qui se contente de chercher les caractères `+` et `%` et laissant les autres inchangés, on peut avoir une écriture plus claire :

```
_load+"locked/stduser.pkg"%0amain+"SCOL.cryopolis.com:Cryopolis"+ffff
ffff+NIL
```

Ce qui donne :

```
...LaunchMachine('$browser$_load+"locked/stduser.pkg"%0amain+"SCOL.cr
yopolis.com:Cryopolis"+ffffffff+NIL CSDMRWK 262144'...
```

Attention, on ne peut utiliser le caractère guillemet (`"`) qu'à la condition d'utiliser l'apostrophe (`'`) comme délimiteur des chaînes de caractères en *'Visual Basic'* ou *'Java Script'* : …`LaunchMachine(` ' $browser…)

- le script lançant la machine SCOL en mode autonome (la page Web lance une machine SCOL qui n'est pas intégrée dans la page, mais qui n'est pas détruite lorsqu'on change de page ou lorsqu'on ferme le browser) :

```
_load "locked/link.pkg"
main "url"
```

On obtiendra alors la chaîne script suivante :

```
_load+"locked/link.pkg"%0amain+"SCOL.cryopolis.com:Cryopolis"
```

Ce qui donne :

```
...LaunchMachine('$browser$_load+"locked/link.pkg"%0amain+"SCOL.cryop
olis.com:Cryopolis" CSDMRWK 262144'...
```

13.4.1.1. Passage de paramètres dans l'url

Si votre site utilise l'architecture DMS (site créé avec ACTIV Shop par exemple), l'url permet également de passer des paramètres à la machines (il s'agit de variables de ressources que la machine SCOL pourra lire grâce à la fonction `_getress`).

La forme de l'url est alors :
`Nom_machine:port_ou_nom_service/ressource1+valeur1/ressource2+valeur2…`

Par exemple, vous voulez passer une variable de ressource 'login' avec la valeur Alice, et une variable de ressource 'authorization_number' avec la valeur 123456, pour le site Cryopolis. L'url associée est :
`SCOL.cryopolis.com:Cryopolis/login+Alice/authorization_number+123456`

La machine SCOL pourra lire les valeurs 'Alice' et '123456' en appelant respectivement `_getress "login"` et `_getress "authorization_number"`.

Remarque : si la valeur contient des caractères spéciaux, comme le `+` ou le `'`, souvenez vous que votre chaîne va être lue au format " `webtostr` " et donc remplacez ces caractères par `%code_ascii_hexadécimal`.

Par ailleurs, si vous avez des caractères espaces, retour chariot, etc, considérez ce qui suit : le programme suppose qu'entre deux caractères " `/` " de l'url, il peut appliquer les fonctions SCOL `webtostr` puis `strextr` pour obtenir une liste (`ressource : :valeur : :nil) : :nil`

Si votre site n'utilise pas l'architecture DMS (ce qui devrait être plutôt rare), vous récupérez toutes ces définitions de ressources sous forme d'une liste `[[S r1] r1]` contenant une ligne par définition, dans la variable " parameters ".

Au lieu de `_getress "login"`, vous pourrez utiliser `switchstr (strextr _getress "parameters") "login"`

13.4.1.2. Utilisation de la fenêtre fournie au composant par le container

La machine SCOL ainsi lancée en composant **activeX** peut utiliser la zone que lui attribue le container en utilisant la fonction suivante :

```
_GETactiveXWindow : fun [Chn I S] ObjWin
```

L'entier est le flag de la fenêtre, la chaîne est le nom de la fenêtre. Cette fonction retourne `nil` si la fenêtre n'est pas disponible, ce qui est le cas si :

- la machine SCOL n'a pas été lancée comme un composant ActiveX ou un plug-in Netscape
- la zone du composant a été définie dans une page web invisible (seulement pour certains browsers)

13.4.2. Interfaçage évolué

Il est intéressant dans certaines applications de faire communiquer une page Web avec le composant SCOL qu'elle contient, en utilisant '*JavaScript*' dans la page Web. Ceci est parfaitement possible en SCOL.

Le lancement du composant est différent sous Netscape, car pour rendre cette communication possible, le composant doit être lancé par Java (et non '*JavaScript*'). La page Web de base est donc la suivante :

```
<body bgcolor="#000000" text="#7DcDe7" link="#FFFF00"
vlink="#FFFF00" alink="#FFFF00" onLoad="Load()">
<OBJECT ID="SCOL"
        NAME="SCOL"
        WIDTH=500
        HEIGHT=400
        align="baseline"
        border="0"
        CLASSID="CLSID:7A96FF35-4937-11D1-8F2C-00609779BDA3">
<EMBED name=SCOL
       type=application/x-SCOL
       border="0"
       width=500
       height=400>
</EMBED>
<SCRIPT LANGUAGE="JavaScript">
<!--
function Message(txt)
{
  selectmsg(txt);
}
//--></SCRIPT>
<applet name="mySCOL" code="SCOLTest.class"
            width=5
        height=5
            mayscript>
</applet>
</OBJECT>

<script LANGUAGE="JavaScript">
<!--
function Load()
{
  if(navigator.appName=='Netscape')
  {
```

```
document.mySCOL.run('$browser$_load+"locked/stduser.pkg"%0amain+"SCOL.cryop
olis.com:Cryopolis"+ffffffff+NIL CSDMRWK 262144',document);
  }
  else
  {
SCOL.LaunchMachine('$browser$_load+"locked/stduser.pkg"%0amain+"SCOL.cryopo
lis.com:Cryopolis"+ffffffff+NIL CSDMRWK 262144',1,0);
  }
}
//-->
</script>

<script language="JavaScript"><!--
function selectmsg(msg)
{
...
}
//--></script>

<script language="VBScript"><!--
Sub SCOL_Message(msg)
  call selectmsg(msg)
end sub
--></script>
```

Dans cet exemple, la fonction `selectmsg` reçoit les messages envoyés par la machine SCOL. Ces messages sont envoyés par la fonction :

`_onX : fun [Comm] I`

Attention, pour une raison qui n'appartient qu'aux développeurs de Netscape, le code javascript traitant le message doit spécifier des url complètes et non relatives, sinon, la machine Java se plante sans avertir. Merci Netscape.

Pour envoyer un message à la machine SCOL, il faut d'abord que celle-ci définisse le canal sensé les recevoir : on utilise pour cette définition la fonction :

`_setX : fun [Chn] Chn`

Il suffit ensuite d'écrire en JavaScript, par exemple :

```
if(navigator.appName=='Netscape')
{
 document.SCOL.SendMessage('_f 1 "abc"');
}
else
{
 SCOL.SendMessage('_f 1 "abc"');
}
```

Ce message sera reçu par le canal défini par la fonction `_setX`. La machine SCOL cherchera alors une fonction `_f` prenant deux arguments, un entier et une chaîne. Le format du message est celui des messages SCOL et donc celui des scripts.

13.4.2.1. Integration ActiveX en mode container

La technologie ActiveX est une technologie Microsoft, disponible uniquement sous Windows. Elle repose sur la notion de composants et de container :

- un composant est une fonctionnalité quelconque, relativement autonome, qui a généralement besoin d'une fenêtre dans laquelle il va pouvoir offrir une interface graphique, et qui peut communiquer avec l'extérieur via une API entrante et sortante (envoi/réception de messages, qui sont en fait des jeux d'appels de fonctions et de callbacks)
- un container est un document (typiquement une fenêtre), dans laquelle on définit un certains nombre de zones (sous-fenêtres), dans lesquelles on lance des composants ActiveX. Une fois ces composants lancés, le container peut communiquer avec eux via l'API du composant.

La technologie SCOL est à la fois composant et container ActiveX. Le mode composant ActiveX est utilisé pour intégrer SCOL dans le browser Internet Explorer (qui est lui-même un container ActiveX). Il permet d'utiliser dans SCOL des fonctionnalités disponibles sous forme de composants ActiveX.

Par exemple, côté client, une page Web (Internet Explorer est également un composant ActiveX), un viewer particulier, … Ceci présente toutefois deux limitations (qui n'existent que si votre service est ouvert au public) :

- la technologie ActiveX n'étant disponible que sur Windows, seuls les visiteurs Windows de votre site accéderont à ces fonctionnalités
- SCOL ne prend pas en charge, pour des raisons de sécurité, le téléchargement de vos composants ActiveX. Vous devez donc vous charger vous-même de cette opération. Cependant, le programme SCOL peut détecter la présence d'un composant ActiveX.

13.4.2.2. interfaçage par socket Bsd (librairie Telnet)

La machine SCOL dispose d'une API Telnet (sockets BSD clientes). Ceci peut être utile côté serveur ou côté client :

- côté serveur, vous pouvez ouvrir une connexion vers un service Tcp-Ip quelconque.
- côté client, vous pouvez également vous connecter à un service Tcp-Ip quelconque. Toutefois, la plupart des firewalls bloquent ce type de connexion, et il n'y a aucun moyen de le contourner. Notons cependant que la librairie Telnet de SCOL est compatible avec les proxies de type SocksHost.

13.4.2.3. interfaçage par socket SCOL

Il est toujours possible de se connecter à une machine SCOL en se faisant passer pour une autre machine SCOL. Ceci simplifie grandement l'interfaçage, en bénéficiant du parsing et de la gestion des canaux déjà intégrés dans une machine SCOL. On utilise alors des sockets Tcp-Ip classiques pour lesquelles on utilise le protocole suivant :

Chaque message est composé de deux parties :

- un header de deux octets codant la taille du corps de message (dans l'ordre octet faible, octet fort)
- un corps de message qui est en fait une ligne de script SCOL (reportez vous au chapitre correspondant pour le détail de la syntaxe).

14. PROGRAMMATION DMS : DISTRIBUTED MODULES SYSTEM.

Ce chapitre décrit l'architecture de programmes appelée DMS. Cette architecture est de type "composants", et a donc pour objectif d'éviter au développeur de tout redévelopper à chaque nouvelle application en lui permettant d'utiliser les composants des autres. Pour cette raison, il n'est pas possible de détailler un exemple complet d'application. Aussi, on se contentera de présenter des exemples de composants.

14.1. Présentation

La technologie SCOL repose sur un langage de programmation intégrant des possibilités de communication internet auquel on associe un certain nombre de librairies graphiques, 3d, multimédia, sql, ... Le premier objectif de la technologie SCOL est donc atteint : mettre entre les mains des développeurs un outil simple et puissant, permettant de gagner en vitesse et en fiabilité de développement. Les difficultés techniques sont réglées par la technologie, le développeur n'a plus qu'à s'intéresser aux vrais problèmes, ceux spécifiques de l'application qu'il développe.

Le second objectif de la technologie est plus ambitieux : la technologie SCOL peut aussi intéresser un public créatif non rompu aux techniques de programmation, c'est-à-dire à un public intégrateur, qui serait capable de récupérer un élément graphique par ci, un bout d'interface, un morceau de programme par là, pour finalement construire une application de type distribué, par exemple un monde virtuel.

Pour cela, il aurait été possible de programmer une sorte de Wizard qui demande les goûts de l'utilisateur, propose quelques choix et construit l'application prête à l'emploi. Cette démarche se serait révélée rapidement limitée et décevante, un tel outil étant incapable de souplesse.

La solution retenue a été de définir une méthode particulière d'utiliser la technologie SCOL. Cette méthode permet d'homogénéiser les développements de programmation en introduisant la notion de "module". Les modules étant distribués (au sens informatique du terme), la méthode de programmation a pour nom DMS : Distributed Modules System.

Un module est un morceau de programme qui effectue une fonction quelconque. Vu de l'extérieur, les modules se présentent tous de la même manière, un peu comme un circuit intégré : le boîtier est le même, les pattes se ressemblent, seuls le nombre, le sens (entrée ou sortie) et la fonction des pattes diffèrent. Pour créer une application, il suffit simplement d'assembler les modules et de tendre des liens entre les pattes. Ceci se fait **à la souris et sans programmation**.

Les modules sont distribués : une partie fonctionne sur une machine appelée 'Serveur', une autre partie fonctionne sur les machines des utilisateurs appelés 'Clients'. Décider ce qui doit être calculé sur le serveur et ce qui doit être calculé sur les clients est un problème informatique complexe, hors de portée du grand-public. Le module masque donc ce problème : ce n'est pas à celui qui assemble les modules de trancher cette question, c'est le développeur d'un module qui se la pose.

Il existe d'autres systèmes basés sur une programmation modulaire. L'originalité de DMS est double :

- l'assemblage des modules se fait en tendant des liens entres les modules, et non par un langage de programmation appelé souvent langage de script. L'usage d'un tel langage, même simple, rend l'utilisation de ces architectures hors de portée du grand public
- les modules sont distribués, mais l'utilisateur n'a pas à s'en soucier. Il n'a pas à définir un assemblage de modules pour le serveur et un autre pour les clients, il ne définit qu'un seul assemblage.

L'architecture DMS n'est pas seulement utile pour le grand public, elle permet également au développeur de gagner beaucoup de temps. Il est facile de développer un module DMS. Transformer une fonction simple en module DMS permet de ne pas avoir à développer des fonctionnalités déjà présentes dans d'autres modules : fichiers de log, fenêtres de console, mots de passe, statistiques, ... Ainsi le DMS évolue continuellement, pour le bien de tous, au fur et à mesure que se développent de nouveaux modules.

Pour développer un module DMS, il faut écrire deux ou trois morceaux de programmes :

- **le module serveur** : programme fonctionnant sur le serveur
- **le module client éventuel** : si une partie du traitement se fait sur les postes clients (interfaces principalement), il faut écrire le programme qui fonctionnera sur les clients
- **l'éditeur du module** : cet éditeur s'intégrera dans l'éditeur des sites DMS (ACTIV SHOP)

Chacun de ces morceaux de programmes utilise une Api qui est détaillée plus loin.

14.2. Définitions

Commençons par quelques définitions :

Site Dms
Application distribuée conforme à l'architecture décrite dans ce document

Client
Logiciel fonctionnant sur l'ordinateur d'un utilisateur d'un site Dms

Serveur
Logiciel permettant la mise en relation des utilisateurs. Pour chaque site Dms, il existe un serveur.

Module
Elément constitutif d'un site Dms. Généralement distribué, il a une partie serveur qui fonctionne côté serveur, et une partie cliente qui est dupliquée sur les postes des clients. Pour simplifier le discours on dira 'module serveur' pour dire 'partie serveur d'un module', et 'module client' pour dire 'partie cliente d'un module'.
De même, on dira 'module client et module serveur associés' pour 'parties clientes et serveur d'un même module'.

User
Notion généralisante d'un utilisateur présent dans le site. Cet utilisateur est soit réel (il correspond alors à un client), soit virtuel (il s'agit d'une entité résidant sur le serveur). Les users se déplacent le long des liens.
A chaque client correspond exactement un User.

Evénement
Signal sortant d'un module. L'événement est soit client (l'événement est produit chez un client), soit serveur (l'événement est produit sur le serveur) : cette localisation est décidée par l'auteur du module. Un User est généralement associé à un tel signal : en fait le signal est le signe du " déplacement " d'un User.

Action
Signal entrant dans un module. L'action est soit cliente (l'action est normalement traitée par un module client), soit serveur (l'action est traitée par le module serveur) : cette localisation est décidée par l'auteur du module.

Message
Message envoyé entre un module serveur et un module client associés (dans un sens ou dans l'autre).

Lien
Association d'un événement d'un module avec une action d'un autre module, éventuellement soumis à condition. On peut attacher un paramètre à un lien. On parlera de 'paramètre du lien'.

Zone
Zone graphique rectangulaire utilisée par un module pour présenter un résultat, une interface, ...

Document
Fenêtre graphique dans laquelle une ou plusieurs zones peuvent être définies. Les documents sont organisés dans une hiérarchie. Cette hiérarchie définit deux types de fenêtres filles : les documents fils " popup " (s'ouvrant par dessus le document père), et les documents fils simples, correspondant à une zone du document père. Un document principal est un document qui n'a pas de document père.
Un site Dms utilise deux documents principaux, le document serveur et le document client.

14.3. Principes

14.3.1. Architecture de modules

L'architecture d'un site Dms est modulaire : un site est constitué d'un nombre variable de modules, connectés entre eux par des liens. Chaque module gère une ou plusieurs fonctionnalités : log, authentification, espace 3d, ... Chaque module peut avoir un fonctionnement distribué : une partie du traitement s'effectue sur le serveur, une autre sur les clients.

Le rôle de l'auteur d'un site Dms est donc de sélectionner un certain nombre de modules et de les assembler en tendant des liens entre eux.

Un des problèmes majeurs dans la réalisation d'une application mettant en relation différents utilisateurs est de concevoir la répartition du traitement entre le serveur et les clients. Ce problème est éminemment technique, et donc hors de portée a priori de celui qui va construire un site Dms. Le problème de la distribution sera donc réglé à l'intérieur d'un module, par celui qui a développé le module.

Dans le cas général, chaque module sera divisé en deux, une partie fonctionnant sur le serveur, une autre sur les clients. La communication entre la partie cliente et la partie serveur d'un même module sera gérée exclusivement par l'auteur du module, en utilisant les techniques classiques de communication incluses dans SCOL.

La communication entre les modules se fera sous forme de liens, et sera donc définie par l'auteur du site.

Les modules possèdent un nom dont les seules contraintes sont que :

- deux modules frères doivent avoir des noms différents (voir plus loin l'**encapsulation** pour comprendre cette notion de modules frères)
- un nom ne doit pas commencer par le caractère '.'

14.3.2. Arborescence de documents

Un module peut à tout moment demander l'utilisation d'une zone d'un document, ou au contraire cesser d'utiliser une zone. Le système gère l'affichage des documents de manière à ce que le document soit visible dès lors qu'au moins une de ses zones est présentement utilisée par un module. Un document ne contenant plus de zone utilisée est détruit, à l'exception du document principal, père de tous les autres, et qui témoigne de la présence de l'application.

14.3.3. Encapsulation

Les modules peuvent être encapsulés : un ensemble de modules peut être regroupé et remplacé par une boîte noire qui les contient. Les modules sont donc en fait organisés dans une arborescence de modules dont les nœuds non-feuilles sont des boîtes noires. Chaque module peut alors tendre des liens vers un module frère, un module fils ou encore vers le module père. La boîte noire se comporte comme un relais transparent : les actions sont directement connectées à des événements, dans les deux directions.

14.3.4. Liens et communication inter-modules

Un lien relie un événement d'un module avec une action d'un autre module. Le nombre de liens affectés à un événement donné ou à une action donnée n'est pas limité. Les liens sont directionnels : de l'événement vers l'action. A chaque lien est associé un paramètre par défaut et/ou une condition.

A tout moment, un module peut déclencher un événement. Le système le convertit, selon les liens, en actions pour d'autres modules. Chaque module concerné par l'action est averti de l'identité de l'émetteur, et peut lui répondre si celui-ci l'a prévu en utilisant un système de " tags ".

On distingue donc deux types de communications inter-modules :

- **par événement** : un événement est envoyé au système, celui-ci le convertit en actions, conformément aux liens tracés par l'auteur du site. L'événement peut être émis par le module serveur ou le module client. L'action peut être reçue par le module serveur ou le module client. Toutefois, le lien client->serveur peut introduire une faiblesse de sécurité : rien ne prouve que le module client fonctionne correctement.
- **par réponse** : lorsqu'un module émet un événement, il peut lui associer un " tag " de réponse, qui est en fait une fonction callback qui attend qu'on lui spécifie un paramètre et/ou une liste de Users. Le module qui reçoit une action associée à cet événement peut alors répondre à ce " tag " en fournissant un paramètre et/ou une liste de Users.

Un événement représente en fait le "déplacement" d'un User, sortant d'un module via une certaine patte. Les liens tracés à partir de cette patte mène le User vers d'autres modules. Ceci ne signifie pas pour autant que le User quitte le module qui a produit l'événement : il y a possibilité d'ubiquité, un User peut être présent dans plusieurs modules à la fois.

Pour cette raison, un événement est défini par les éléments suivants :

- **un User** : celui qui se déplace dans le graphe de modules
- **un paramètre** : une chaîne de caractère quelconque. Si celle-ci vaut `nil`, le paramètre est remplacé par le premier paramètre par défaut non `nil` associé aux liens que l'événement suit.
- **une liste de Users** relatifs à l'événement
- **un " tag "** de réponse

Chacun de ces éléments est facultatif et peut être remplacé par `nil`. Cependant, il est extrêmement rare que l'argument User soit `nil`, puisqu' un événement est justement le déplacement d'un User.

Lorsqu'un événement est produit par un module client, le User associé est implicitement celui qui correspond au module client.

Lorsqu'un événement aboutit à une action située normalement sur un module client, le routage se fait en fonction du User associé :

- si le User est réel (c'est-à-dire correspond à un client), l'action est traitée par le module du client associé
- si le User est virtuel, l'action est traitée par le module serveur.

Ainsi, le paramètre User d'un événement permet-il de régler tous les problèmes de routage.

14.3.5. Activation dynamique

Créer un site consiste à assembler un ensemble de modules. Chaque machine (clients ou serveur) abrite une copie approximative de ce site : cela signifie que certains modules ne sont pas représentés sur chaque machine. Par exemple seul le module de l'espace 3d ou se trouve un utilisateur est représenté sur sa machine ; par contre tous les modules espace 3d du site sont représentés sur le serveur. Il y a derrière cette observation la question de l'activation dynamique des modules : alors que tous les modules sont actifs côté serveur, seul quelques modules sont actifs côté client : le nombre et la nature des modules clients actifs varie en fonction du temps et du client.

C'est en fait le module serveur qui déclenche l'activation du module client chez un client donné. Le système gère pour chaque module serveur la liste des modules clients activés, ce qui lui permet notamment d'effectuer un filtre sur les messages, ce qui garantit la sécurité des échanges de messages. L'activation n'est donc pas automatique : ce n'est pas parce qu'un événement est lié à une action d'un module que ce module sera créé lorsque l'événement se produira. Si l'événement se produit alors que le module recevant l'action n'est pas créé, l'événement est simplement ignoré.

Pour activer un module, un certain nombre de données doivent être transmises au client, et celui-ci doit si c'est nécessaire télécharger un certain nombre de fichiers (notamment des fichiers sources SCOL décrivant le fonctionnement du module client). Activer un module sur un client est donc une opération complexe qui peut nécessiter un certain temps. Durant ce temps, le module client est en " sommeil " et bufferise les messages et actions qu'il reçoit en attendant de pouvoir être réellement lancé : une fois que tous les fichiers nécessaires sont présents, le module client est compilé, lancé, et tous les messages bufferisés sont traités.

14.3.6. Users et UserInstances

On a vu que les Users sont les éléments mobiles de l'architecture DMS : ils se déplacent le long du graphe de module, lors des événements. Un module reçoit donc un flux de Users.

Lorsque le module le juge utile, il peut définir une structure appelée `UserInstance`. Pour un module donné, on ne peut définir qu'un `UserInstance` par User. L'objet `UserInstance` est un objet :

- associé à un User,
- distribué et synchronisé entre les parties clientes et serveur d'un module, et offrant des possibilités de communication extrêmement pratiques entre ces différentes parties.

14.3.7. L'éditeur de sites ACTIV SHOP

L'éditeur de sites (dont le nom commercial est ACTIV SHOP) est l'outil associé à l'architecture DMS. Il permet de :

- sélectionner les modules à intégrer au site : créer, supprimer
- définir les documents serveur et client
- définir les liens entre les modules
- affecter les zones au modules : indiquer dans quelle zone afficher la 3d, un bouton, une image, ...
- lancer les éditeurs de module

En effet, chaque module contient normalement un éditeur qui permet de le paramétrer. Par exemple :

- définir un espace 3d pour un module de gestion d'espace 3d

- définir les textes d'un module bandeau d'affichage

On distinguera bien les deux types d'éditeur : éditeur de site et éditeurs de modules. L'éditeur de module est normalement écrit par le développeur du module.

14.4. Téléchargement de ressources

Le serveur joue également le rôle de distributeur de ressources. Une ressource correspond à un ensemble quelconque d'octets. Une ressource est nommée et appartient à un module qui est son propriétaire. Les ressources sont les données que les clients vont pouvoir télécharger : fichiers contenant le programme du module client, fichiers graphiques, 3d ou son, ... Il est important que chaque module enregistre les ressources dont le client peut avoir besoin, car c'est ce qui garantit les possibilités de mise à jour automatique et de code mobile : dans le cas contraire, seuls les clients possédant déjà ces fichiers pourront fonctionner normalement. Ceci peut être soit un oubli, soit une omission délibérée pour sélectionner les clients. Lorsqu'on développe un module, il est important de vérifier qu'en se connectant sur le site, un client "vierge" téléchargera bien les données clientes propres au module.

Une ressource est normalement accessible à tout client dont le module client correspondant au module serveur propriétaire est activé. Cependant, la ressource peut être sécurisée avec un accès restreint aux seuls clients autorisés.

Côté serveur, la ressource peut être présente dans la mémoire du serveur ou stockée sur le disque. Dans le premier cas seulement elle pourra être compressée avant le téléchargement par le client. Dans le deuxième cas, les données seront transférées directement : il est donc recommandé que le fichier soit déjà compressé (fichier graphique jpeg par exemple)..

Côté client, la ressource peut être stockée dans le cache sous un nom donné qui permettra lors d'un usage ultérieur de déterminer si la ressource doit être téléchargée ou si elle est déjà présente. La ressource peut également être fournie directement au client, sans passer par un stockage sur disque.

Le serveur calcule la signature des ressources qu'on lui soumet. Cette signature est transmise au module client lors de sa création. Le module client vérifie la signature du fichier portant le nom de la ressource, et si elle ne correspond pas, télécharge de nouveau la ressource. Vérifier qu'un client possède bien un fichier donné se fait donc en comparant la signature du contenu, et non pas en regardant la date de création, car ce dernier procédé est bien moins fiable.

Un module serveur peut :

- enregistrer une ressource en spécifiant son nom
- désenregistrer un ressource en spécifiant son nom
- désenregistrer toutes les ressources
- autoriser un client à accéder à une ressource

Un module client peut :

- demander le téléchargement d'une ressource en connaissant son nom, et choisir de passer par le cache disque ou non
- arrêter le téléchargement d'une ressource
- arrêter tous les téléchargements qu'il a demandés.

Un client peut également transférer une ressource au serveur, il s'agit du mécanisme d'upload.

Attention : lorsqu'on développe un module et que l'on travaille sur la partie cliente, il ne faut pas oublier de relancer le serveur pour que les modifications apportées au code client soient prises en compte. En effet, le serveur calcule lors de son démarrage la signature des ressources. Si vous modifiez une ressource sans relancer le serveur, celui-ci garde en mémoire la signature de l'ancienne ressource. Si vous lancez alors un client, le serveur lui retransmettra l'ancienne ressource qui sera

placée dans la partition cache. Celle-ci masquera la nouvelle ressource, et vous vous demanderez pourquoi vos modifications ne sont pas prises en compte.

14.5. Fichiers de définition d'un site DMS

La définition d'un site DMS se fait au travers de deux types de fichiers.

Les fichiers Dmc (Distributed Modules Class) définissent une classe de module : par exemple un module de gestion de log, un module de gestion 3d, un module d'interface texte, ... Un fichier Dmc contient la liste de fichiers utilisés par la classe du module --typiquement les morceaux de programme-- qui seront enregistrés automatiquement comme ressources sur le serveur, puis les scripts de lancement du module, un pour le serveur, un pour le client, un pour l'éditeur.

Le fichier Dms contient la définition d'un site :

- le graphe de modules (arbre de modules et liens inter-modules)
- la définition des documents clients et serveur
- le paramétrage de chaque module.

Il est utile de préciser les origines des différents éléments :

- le fichier Dmc est créé par le développeur du module
- le fichier Dms est créé par l'éditeur de site (ACTIV SHOP)
- les blocs de définition d'un module (bloc 'dmi' en tête) sont créés par l'éditeur du module

14.5.1. Fichier dmc : distributed modules class

Un fichier dmc décrit une classe de modules.

- `name [nom de classe]` : nom de la classe (pour l'éditeur)
- `register [fichier 1] ... [fichier n]` : liste des fichiers à enregistrer en les chargeant en mémoire
- `registerF [fichier 1] ... [fichier n]` : liste des fichiers à enregistrer sans les charger en mémoire
- `serverNeeded [fichier 1] ... [fichier n]` : liste des fichiers requis pour lancer le serveur
- `serverLoad [fichier 1] ... [fichier n]` : liste des fichiers à compiler successivement pour lancer le serveur clientNeeded [fichier 1] ... [fichier n] : liste des fichiers requis pour lancer le client
- `clientLoad [fichier 1] ... [fichier n]` : liste des fichiers à compiler successivement pour lancer le client
- `editorNeeded [fichier 1] ... [fichier n]` : liste des fichiers requis pour lancer l'éditeur du module
- `editorLoad [fichier 1] ... [fichier n]` : liste des fichiers à compiler successivement pour lancer l'éditeur
- `bitmap [fichier]` : nom du fichier contenant l'icone par défaut à utiliser par l'éditeur de site.
- `helpFile [fichier aide]` : fichier d'aide pour l'éditeur

Tous les noms de fichiers sont :

- soit absolus : par exemple `'dms/admin/log/log.pkg'`
- soit relatifs descendants par rapport au répertoire du fichier dmc. Ils commencent alors par './' : par exemple `'./log.pkg'`. La séquence `'../'` n'est pas reconnue.

Les fichiers des lignes `'…Load'` n'ont pas besoin d'être réécrits dans les lignes `'…Needed'` : le système considère qu'ils sont forcément nécessaires au lancement du module (puisqu'ils doivent être compilés).

Pour déterminer la liste des fichiers utiles au site (en vue de dupliquer par exemple le site sur un autre serveur), le système concatène les listes `register`, `registerF`, `serverNeeded`, `serverLoad`, `clientNeeded` et `clientLoad`.

Décrivons le mécanisme de démarrage d'un module serveur. Lorsqu'un module serveur est créé, le système vérifie la présence des fichiers `'serverNeeded'`. Puis il crée un canal unplugged héritant de l'API serveur, compile successivement dans ce canal les fichiers de la ligne `serverLoad`, puis exécute la fonction `IniDMI` (voir plus loin).

Précisons le mécanisme de démarrage d'un module client. Lorsque le module est activé sur un poste client, celui-ci commence par s'assurer qu'il possède le fichier Dmc. Si ce n'est pas le cas, il le télécharge depuis le serveur. Ce fichier n'a pas à être déclaré dans la liste des ressources (lignes `'register"` et `'registerF'`), car ceci est fait automatiquement pour tous les modules présents sur le serveur.

Puis, le client s'assure qu'il possède tous les fichiers de la ligne `'clientNeeded'`. S'il lui manque des fichiers, il demande au serveur de les lui fournir. Ceci n'est possible que si ces fichiers apparaissent sur la ligne `'register'` ou `'registerF'`. Si un seul fichier est manquant, une boite de dialogue apparaît sur le client et celui-ci s'interrompt.

Lorsque tous les fichiers requis sont présents, le module client est créé de la manière suivante : un canal unplugged est créé, héritant de l'API client, puis les fichiers de la ligne `clientLoad` sont compilés successivement, puis la fonction `IniDMI` est appelée (voir plus loin).

Les fichiers placés sur la ligne `'register'` sont traités de la manière suivante :

- ils sont lus, puis compressé (fonction **zip**)
- puis il sont conservés dans la mémoire de la machine SCOL
- ils sont transmis au client qui les demande

Les fichiers placés sur la ligne `'registerF'` sont traités différemment :

- ils sont lus, mais non compressés
- ils ne sont pas conservés en mémoire, mais sur le disque
- lorsqu'un client les demande, ils sont relus sur le disque du serveur.

L'unique avantage des fichiers `'registerF'` est de ne pas encombrer la mémoire du serveur. C'est pourquoi le module 3d place les textures sur cette ligne. L'absence de compression n'est pas pénalisant car la plupart des textures sont au format jpeg, qui est déjà très bien compressé.

Pour ce qui est de l'éditeur, pour lancer l'éditeur du module, l'éditeur de site vérifie que les fichiers `'editorNeeded'` sont présents. Si un seul manque, une boîte de dialogue apparaît pour indiquer que l'éditeur ne peut être lancé. Si tous les fichiers sont présents, l'éditeur crée un canal unplugged héritant de l'API éditeur, y compile successivement les fichiers de la ligne `'editorLoad'`, puis la fonction `'IniEditor'` (voir plus loin).

Dans la précédente version, on trouvait les lignes `serverScript`, `clientScript` et `editorScript` qui définissaient le script de lancement des parties serveur, client et éditeur. Ces scripts étaient toujours une série de `_load`. On a donc remplacé ce système par les lignes `serverLoad`, `clientLoad` et `editorLoad` qui sont plus simples à utiliser et offre la possibilité de définir des chemins relatifs. Toutefois, la compatibilité ascendante est assurée.

Le fichier d'aide spécifié sur la ligne `'helpFile'` peut être au format HTML (extensions .htm ou .html) ou au format texte (toutes autres extensions). Les fichiers au format HTML sont considérés

comme localisés, c'est à dire que le fichier recherché sera le fichier correspondant à la langue paramétrée dans le SCOL Voy@ger et par défaut l'anglais.

Exemple : Soit la ligne `helpFile ./help/help.htm` spécifiée dans le fichier dmc, le fichier d'aide ouvert par le bouton help de l'éditeur sera './help/help.french.htm' pour un SCOL Voy@ger configuré en français.

14.5.2. Fichier Dms

Le fichier Dms représente une arborescence de modules. Chaque nœud contient un nombre quelconque de blocs de définitions nommés, et deux de ces blocs ne peuvent avoir le même nom. Un bloc de définitions est composé d'une liste de lignes '`nom {valeur1 {valeur2 … {valeurN}…}}`'. Les valeurs sont des chaînes d'octets quelconques.

Le nœud principal du site est en fait la "boîte noire" qui contient tout le site. Il contient les définitions globales du site (nom, port, documents, …).

La syntaxe du fichier dms est la suivante (le fichier est au format `strextr`) :

```
Module : :=
module nom numéro_de_serveur
(Definition)*
(Module)*
endmod

Definition : :=
def nom
(Ligne)*
enddef

Ligne : :=
> nom (valeur)*
```

Chaque module feuille contient :

- un bloc de définitions '**dmi'** qui contient :
  - la classe (c'est-à-dire le nom du fichier dmc),
  - les événements,
  - les actions,
  - les zones
  - la liste des fichiers qu'il utilise et qui font donc partie du site (images, textures, 3d, son, ..), ceci en vue de "dupliquer" facilement un site d'un serveur vers un autre.
- un bloc de définitions '**link'** qui contient :
  - les liens sortants du module
- un bloc de définitions '**zone'** qui contient :
  - les correspondance `zone->document`

Chaque module non feuille contient :

- un bloc de définitions '**dmi'** qui contient :
  - les pattes entrantes ('`in`')
  - les pattes sortantes ('`out`')
- un bloc de définitions '**link'** qui contient :
  - les liens sortant d'une des pattes

Le module global (nœud principal) contient :

- un bloc de définitions '**def'** qui contient :
  - les définitions globales du site :

```
- name
- port
- timeout
```

- un bloc de définitions '**docserver'** qui contient :
  - la description du document serveur
- un bloc de définitions '**docclient'** qui contient :
  - la description du document client (anciennement fichier Scc)

Détaillons les champs de chaque type de bloc de définition.

14.5.2.1. Bloc de définition 'dmi' : distributed module instance

Un bloc de définition **dmi** décrit les paramètres de base d'une instance de module, ou d'une boîte noire.

Dans le cas d'une instance de module, on trouve :

- `name [nom de l'instance]` : nom de l'instance
- `class [fichier dmc]` : fichier de classe
- `event [name]` : événement produit sur le serveur
- `eventC [name]` : événement produit sur le client
- `action [name]` : action sur le serveur
- `actionC [name]` : action normalement sur le client
- `zone [nom de zone module]` : nom alias d'une zone utilisée par le module (pour l'éditeur)
- `zoneC [nom de zone module]` : nom alias d'une zone utilisée par le module client (pour l'éditeur)
- `register [fichier 1] ... [fichier n]` : liste des fichiers à enregistrer en les chargeant en mémoire
- `registerF [fichier 1] ... [fichier n]` : liste des fichiers à enregistrer sans les charger en mémoire
- `clientNeeded [fichier 1] ... [fichier n]` : liste des fichiers spécifiques à l'instance et qui doivent être téléchargés par le client avant le lancement du module
- `serverNeeded [fichier 1] ... [fichier n]` : liste des fichiers utiles au serveur, en vue d'une duplication du site sur une autre machine.
- `bitmap [fichier bitmap]` : bitmap à utiliser dans l'éditeur

Les noms de fichiers sont ici absolus.

Les champs définis ci-dessus sont les champs standard. Certains modules définiront des champs supplémentaires qui leur sont propres : textes des annonces pour un bandeau d'affichage, adresse d'un annuaire pour un module d'enregistrement automatique. Mais il sera plus élégant de placer ces données spécifiques dans des blocs de définition différents.

Les événements et les actions sont définis soit sur le serveur soit sur le client, d'où les lignes `event/eventC`, `action/actionC`. Décider si un événement (resp. une action) doit être défini dans la ligne `event` ou `eventC` (resp. `action` ou `actionC`) est très simple : tout dépend du module qui provoque l'événement (resp. qui traite l'action), serveur ou client.

Les lignes `register` et `registerF` doivent contenir les fichiers spécifiques de l'instance. Il est inutile de re-déclarer ici les fichiers présents dans le Dmc. On fera sur les fichiers `'register'` et `'registerF'` les mêmes remarques que pour les fichiers Dmc.

La ligne `serverNeeded` permet de déterminer les fichiers utiles au site, et qui ne sont pas déjà dans les lignes `register` et `registerF`.

Pour une boîte noire, on trouvera dans le bloc de définition dmi les champs suivants :

- `name [nom]` : nom de la boîte noire
- `in [name]` : patte entrante
- `out [name]` : patte sortante.

14.5.2.2. Bloc de définition 'link'

Le bloc de définition '**link'** contient uniquement des lignes de la forme suivante :

`[event] [destination] [action] [param] [reply] [condition]` : description d'un lien attaché à un événement du module. Le module destinataire est définit par le champ 'destination' de la manière suivante :

- `..` : père
- `.name` : fils
- `name` : frère

Le paramètre `reply` est ici pour compatibilité ascendante.

Les conditions des liens sont une chaîne destinée à être exploitée par la fonction `strextr`. Chaque ligne correspond à une condition d'activation : il y a un OU logique entre les lignes.

Chaque ligne est composée d'une liste de conditions de base. Chaque condition correspond à un ou plusieurs mots de la ligne. Toutes les conditions de base d'une ligne doivent être remplies : il y a un ET logique entre chaque condition de base.

Le premier mot d'une condition de base donne le type de condition, les suivants donnent les arguments. Il y a neufs conditions :

- `!` : condition inverse du reste de la ligne
- `login` : le login vaut-il une certaine valeur (1 paramètre)?
- `notlogin` : le login est-il différent d'une certaine valeur (1 paramètre)?
- `ip` : l'adresse ip vaut-elle une certaine valeur (1 paramètre)?
- `notip` : l'adresse ip est-elle différente d'une certaine valeur (1 paramètre)?
- `item` : l'utilisateur possède-t-il un certain objet (le paramètre est la référence de l'objet) ?
- `noitem` : l'utilisateur ne possède-t-il pas un certain objet (le paramètre est la référence de l'objet) ?
- `items` : l'utilisateur ne possède-t-il pas un certain objet en une certaine quantité (2 paramètres : référence de l'objet et quantité) ?
- `activeX` : le client utilise SCOL en mode composant ActiveX (0 paramètre). Dans ce mode, typiquement, il ne faut pas lancer le browser depuis SCOL, car cela changerait la page courante (celle dans laquelle SCOL fonctionne), et détruirait donc le client. Il faut plutôt que la page Web soit programmée ('*javascript/vbscript*') pour ouvrir une nouvelle frame.

14.5.2.3. Bloc de définition 'zone'

Il est composé de lignes de la forme :

`zoneS [dmi name] [tree zone name]` : correspondance entre le nom d'une zone définie dans le bloc dmi et une zone définie dans l'arbre des documents du fichier dms

`zoneC [dmi name] [tree zone name]` : idem pour une zone utilisée par le module client

14.5.2.4. Bloc de définition 'def'

On trouve les champs suivants :

- `name [nom]` : nom du site
- `port [numéro]` : numéro de port serveur
- `timeout [temps en secondes]` : temps maximal de réponse d'un client à un signal

Le timeout permet de détecter les clients qui se sont déconnectés de manière accidentelle. Le protocole Tcp-Ip est ainsi fait qu'il peut s'écouler plusieurs minutes avant qu'un serveur ne détecte la disparition d'un client. Ici, un signal est envoyé à intervalle régulier à tous les clients. Les clients doivent simplement répondre à ce signal par un autre signal. Lorsque le serveur envoie un signal, il vérifie qu'il a bien reçu le signal précédent, sinon il déconnecte lui-même le client.

14.5.2.5. Blocs de définition 'docclient' et 'docserver'

Ces blocs de définition ont la même syntaxe et définissent respectivement les documents client et serveur.

`doc [nom] [type] [resizeFlag] [x1] [y1] [x2] [y2] [w] [h] [bitmap] [bitmapFlag] [color]` : définition d'un document, avec éventuellement une image de fond de document)
`zone [nom] [resizeFlag] [x1] [y1] [x2] [y2] [w] [h] [color]` : définition d'une zone du document précédemment défini)
`... récursion`

`enddoc` : fin de définition du document (obligatoire)

Les valeurs de 'type' sont :

- 0 : indique que le document est un sous-document interne
- DOCpopup (1) : indique que le document est popup par rapport à son père.

Les valeurs de `'resizeFlag'` sont :

- `ZONE_LW_FLEX` (1): la marge gauche entre la zone et son père est flexible.
- `ZONE_MW_FLEX` (2): la largeur de la zone est flexible.
- `ZONE_RW_FLEX` (4): la marge droite entre la zone et son père est flexible.
- `ZONE_LH_FLEX` (8): la marge haute entre la zone et son père est flexible.
- `ZONE_MH_FLEX` (16): la hauteur de la zone est flexible.
- `ZONE_RH_FLEX` (32): la marge basse entre la zone et son père est flexible.

Dans la taille initiale du document, x1 est la distance à gauche de la zone, w la largeur de la zone, x2 la distance à droite de la zone. De même y1 est la distance en haut de la zone, h la hauteur de la zone, y2 la distance en bas de la zone.

Les valeurs de 'bitmapFlag' sont :

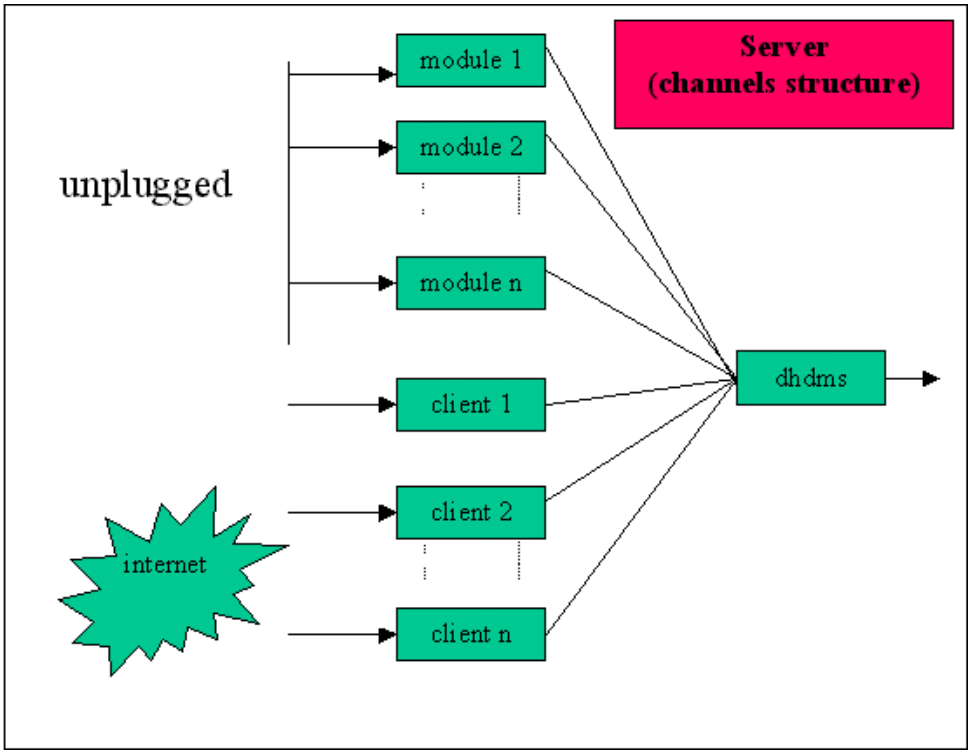
- 0 : indique que le bitmap de fond de document est centré
- `DOCtiled` (4) : indique que le bitmap de fond de document doit être répété en mosaïque.
- `DOCstretched` (8) : indique que le bitmap de fond de document doit être étiré pour entrer dans le document

Un document principal doit être défini pour chaque bloc de définition : il s'agit respectivement des documents '*client*' et '*server*'.

Pour faire apparaître la barre de download, une zone '*download*' doit être définie dans le document '*client*'.

14.6. API

Il faut savoir que le système DMS crée un canal unplugged pour chaque module, que ce soit sur le serveur, le client ou bien l'éditeur. Ce canal hérite des API décrites plus loin. De ce fait la communication entre modules se fait toujours au travers de l'API.



14.6.1. API serveur

14.6.1.1. Structures

- `DMI` : instance d'un module
- `CLIENT` : client
- `User` : user
- `UserI` : user instance
- `Item` : objet de l'inventaire.

14.6.1.2. Variables de l'API

- `this DMI` : à utiliser en tête de plusieurs fonctions de l'API
- `DMSsender CLIENT` : indique l'émetteur d'un message intra-modules
- `DMSname S` : nom du site
- `DMSwin ObjWin` : fenêtre de base du serveur

14.6.1.3. Fonctions de l'API

➢ **Modules :**

S _DMSgetName module DMI
Retourne le nom court d'un module

S _DMSgetClass module DMI
Retourne le nom du fichier dmc d'un module

[[S r1] r1] _DMSgetDef module DMI nom_du_bloc S
retourne un bloc de définition associé à un module

I _DMSupdateDef module DMI nom_du_bloc S données [[S r1] r1]
Redéfinit un bloc de définitions (n'effectue pas de sauvegarde sur disque)

I _DEFsave
Sauvegarde le site.

I _DMScreateClientDMI module DMI client CLIENT param S
Crée l'instance cliente. Retourne 1 si succès. 0 si échec (possible uniquement si l'instance cliente est déjà créée).

I _DMSdelClientDMI module DMI client CLIENT
Détruit l'instance cliente

I _DMSsend module DMI client CLIENT message Comm
envoie un message vers l'instance du même module sur un certain client. Pour que ce message soit interprété, il suffit qu'il existe dans le module client une fonction ayant le même nom, précédé de deux underscores.

➢ **Messages inter-modules : déclenchement d'événements, envois directs :**

I _DMSevent this DMI concerning CLIENT event S param S reply S
active un événement en précisant un paramètre et une réponse. Si le paramètre vaut `nil`, c'est le paramètre par défaut qui est utilisé. Le champ `reply` devrait être laissé à `nil`.

I _DMSeventTag module user User event S param S others [User r1] [callback `fun[param S others [User r1]] I flag timeout]`
Provoque un événement accompagné éventuellement d'un tag. Le flag est inutilisé.

I _DMStagKeepAlive tag Tag
Indique que le tag doit être conservé même s'il ne lui est pas répondu immédiatement.

I _DMStagForget tag Tag
Indique que le tag peut-être oublié.

I _DMSreplyTag tag Tag param S others [User r1] holdon I
Répond au tag en lui passant un paramètre, et en indiquant si le tag doit être détruit (`holdon=0`) ou si l'on compte s'en resservir (`holdon=1`).

I _DMSdefineActions module DMI liste_actions [[S fun [module_émetteur DMI user User action S param S others [User r1] tag Tag] I] r1]
Définit une liste d'actions associées à des callbacks. Cette définition est incrémentale : on peut l'appeler plusieurs fois, et n'importe quand.

I _DMSremoveActions module DMI actions [S r1]
Supprime des actions de la liste.

I _DMSregister (module DMI) (cb_logout fun [CLIENT] I) (cb_deleteclient fun [CLIENT] I) (cb_beforeclose fun [] I)
I _DMSregisterDMI (module DMI) (cb_action fun [from DMI concerning CLIENT action S param S reply S] I) (cb_deleteclient fun [CLIENT] I) (cb_beforeclose fun [] I)
enregistre les callbacks du module.

- La fonction `logout` est appelée lorsqu'un client se déconnecte. Le module doit alors supprimer toute référence vers ce client.
- La fonction `delete` est appelée lorsqu'un module client est détruit : soit le module s'est détruit lui-même, soit le client s'est déconnecté.
- La fonction `beforeclose` est appelée avant que le module serveur ne soit fermé, c'est-à-dire avant la fermeture du serveur.
- La callback action recevra tous les messages actions reçus par le module. La nouvelle fonction `_DMSdefineActions` rend cette callback obsolète.

➢ **Clients :**

Les clients sont définis par une structure CLIENT qui contient principalement :

- un login (qui doit être unique)
- la référence vers le User qui a été créé pour représenter le client.
- ce User est défini par un numéro Id qui restera unique et constant tout au long de la connexion.

User CtoU client CLIENT
Retourne le user associé au client.

I _DMSdelClient client CLIENT
provoque la déconnexion d'un client

S _DMSgetLogin client CLIENT
retourne le login d'un client

S _DMSsetLogin client CLIENT login S
modifie le login d'un client

CLIENT _DMSbyLogin login S
trouve un client en fonction de son login

CLIENT _DMSbyLoginI login S
trouve un client en fonction de son login sans tenir compte de la casse.

I _DMSclientAlive client CLIENT
retourne 1 si le client est encore en vie, 0 sinon : un module ne doit pas conserver de référence vers un client qui n'existe plus.

S _DMSgetIP client CLIENT
retourne l'adresse IP d'un client. Chaque client possède une liste de variables, appelées variables de ressources. L'API permet de définir la valeur de n'importe quelle variable.

S _DMSgetRess client CLIENT ressource S

retourne une variable de ressource associée à un client

`S _DMSsetRess client CLIENT ressource S val S`
définit une variable de ressource associée à un client

- **Users :**

Les User sont les entités mobiles dans le graphe. Elles sont de plusieurs types :

- client/virtuel : correspond soit à un client, soit à un résidant
- global/local : défini pour tout le monde/seulement chez un client donné

Les Users se déplacent dans le graphe. Les services sont simples : création/destruction/lecture des données

Chaque User possède une liste d'items (elle était précédemment gérée au niveau de la structure CLIENT, les anciennes fonctions restent valables). C'est un "inventaire" contenant des objets définis par une référence, un nom en clair, une quantité et une date.

`User UcreateUser client CLIENT`
Crée un user virtuel global.

`I UgetId user User`
Retourne l'Id

`CLIENT UtoC user User`
Retourne le client associé au user (`nil` si le user est virtuel).

`I UgetFlag user User`
Retourne le flag associé au user.

`Item _ITEMcreate référence S nom_en_clair S quantité I date I`
Crée un item

`S _ITEMref item Item`
Retourne la référence

`S _ITEMname Item`
Retourne le nom en clair

`I _ITEMquantity Item`
Retourne la quantité

`I UaddItem user item`
Ajoute un item à un user

`I UsubItem user référence quantité`
Supprime une certaine quantité d'item de l'inventaire d'un user.
Retourne la quantité de l'item qu'il y avait avant cette suppression (0 si l'item n'était pas dans l'inventaire).

`Item UfindItem user chaîne`
Recherche un item dans l'inventaire d'un user

`I UclearItem user`
Supprime tous les items d'un user

➢ **UserInstances :**

Les UserInstances sont un service offert à chaque module pour définir un objet lié à un User et fournir des notions de distribution, de communication et de sécurité.
Les UserInstances sont créées au niveau serveur ou au niveau client, mais un client ne peut créer que des instances locales. On crée les instances en spécifiant :

- un module
- un user
- une classe
- des paramètres
- la visibilité

La notion de visibilité est importante : une instance globale n'est diffusée que sur les clients dont l'instance associée à ce client (c'est-à-dire l'instance associée au User correspondant au client) la 'voit'.

Il y a pour l'instant un seul type de visibilité : par arbre. Chaque instance est placée dans un arbre. La visibilité est alors définie par un chemin dans un arbre (la liste des noms des nœuds en partant du sommet), et un flag de commutativité. La règle de visibilité est alors la suivante.

- une instance "voit" toutes les instances qui sont dans son sous-arbre (même nœud ou descendance)
- si une instance a mis son flag de commutativité à 1, toute instance qu'elle voit la voit également.

De même il est possible d'isoler une instance.

Les services offerts pour les instances globales sont :

- création/destruction automatique d'une instance cliente en fonction des règles de visibilité
- transmission de messages entre ces instances clientes et serveur
- synchronisation de la modification de la classe et des paramètres

La visibilité est définie par le type **: Typedef Visibility**

- arbre `[[S r1] flag]`
- isolé (`nil`)

Visibility treeNew rights [S r1] commut I
Retourne un objet Visibilité par arbre, en spécifiant la position dans l'arbre (liste des noms des nœuds en commençant par le sommet) et flag de commutativité.

`[UserI r1] Ulist module DMI`
Retourne la liste des userI associés à un module

`UserI UcreateUI module DMI user User class S parameters [[S r1] r1] visibility Visibility`
Crée une instance utilisateur. `Nil` comme paramètre de visibilité indique une visibilité nulle. On utilisera typiquement `'treeNew nil nil'` pour définir une visibilité maximale (instance au sommet de l'arbre)

`User UgetUser userI UserI`
Retourne le user associé

`DMI UgetLocation userI UserI`
Retourne la location associée

`[[S r1]r1] UgetParams userI UserI`
Retourne les paramètres associés

[S r1] UgetParam userI UserI champ S
Retourne la valeur du champ

UserI UgetUserI module DMI user User
Retourne le userInstance associé à un module et à un user.

I Udelete userI UserI
Détruit une instance

I UchgClass userI UserI classe S params [[S r1] r1]
Change la classe d'un UserI. Ceci sera transmis aux clients qui voient l'instance, et y provoquera l'appel de la callback définie par `UcbChanged`.

I UsetVisibility userI UserI visibilité Visibility
Change les droits d'un UserI. Cela entraîne en général un certain nombre d'ordre de création/destructions de UserI sur les modules clients.

I UsetParams userI UserI params [[S r1] r1]
Change tous les paramètres. Ceci sera transmis aux clients qui voient l'instance, et y provoquera l'appel de la callback définie par `UcbChanged`.

I UsetParam userI UserI champ S param [S r1]
Change un champ. Ceci sera transmis aux clients qui voient l'instance, et y provoquera l'appel de la callback définie par `UcbChanged`.

UserI UcbClientDestroyed ui UserI callback fun [UserI CLIENT] I
Définit une callback pour être informé de la destruction d'une instance chez un client particulier.

UserI UcbDelete ui UserI callback fun [UserI] I
Définit une callback pour être informé de la destruction d'une instance. Sur le serveur, cela se produit généralement lorsqu'un client se déconnecte : ses userI sont détruits.

UserI UcbMessage ui UserI liste_de_messages [[S callback fun [UserI CLIENT S S] I] r1]
Définit des callbacks sur des réceptions de messages. Ces callbacks sont simplement concaténées à la liste présente et masquent éventuellement des définitions précédentes

UserI UremoveMessage ui UserI message S
Supprime une callback sur un message.

I UsendMessage ui UserI client CLIENT action S param S
Envoie un message à un client. Si client vaut `nil`, le message est envoyé à toutes les instances clientes.

- **Gestion des zones :**

[ObjWin I I I I] _DMSgetZone (this DMI, zone S, conflict fun [zone S] I, resize fun [coord [win ObjWin x I y I w I h I] zone S] I, destroy fun [zone S] I)
demande une zone (le nom à passer est celui de la zone définie dans le bloc de définitions dmi)
la callback `conflict` est activée lorsque la zone est demandée par un autre module. La callback `resize` est activée lorsque la zone a changé de taille. La fonction retourne `nil` si la zone n'est pas associée, et un tuple (fenêtre mère, x, y, largeur, hauteur) dans le cas contraire.

I _DMSreleaseZone (this DMI, zone S)
libère une zone (le nom à passer est celui de la zone définie dans le bloc de définitions dmi)

➢ **Gestion des documents téléchargeables par les clients :**

I _RSregister (this DMI, name S, type I, document S)
Enregistre le document sous un certain nom, avec un certain propriétaire `this`. Si le type vaut 0, le paramètre document est le document lui-même. Si le type vaut 1, le paramètre document est le nom du fichier document. Les transferts se feront de fichier à mémoire, sans compression.
Retourne 0 si Ok, -1 si erreur (document vide ou déjà enregistré).

I _RSregistersafe (this DMI, name S, type I, document S)
Idem, mais le document ne sera accessible qu'aux clients autorisés.

I _RSregisterfiles(this DMI, files [S r1], type I)
Enregistre une liste de fichiers : le nom de chaque document est le nom du fichier.

I _RSunregister (this DMI, name S)
Retire un document

I _RSallowClient (this DMI, client CLIENT, name S)
autorise un client à downloader un document enregistré à l'aide de la fonction `_RSregistersafe`

I _DMScbUpload module DMI callback fun [CLIENT S S] I
Définit une callback de réception de documents (fonction `_DMSupload` de l'Api client). La callback est appelée lorsque la réception est complète avec en arguments : client émetteur, nom du document, contenu.

➢ **Fonctions de localisation :**

L'architecture Dms intègre un kit de localisation permettant l'affichage de messages textes dans différentes langues. Les messages sont stockés dans des fichiers de ressources.

Pour chaque module, un fichier de ressource par langue doit être créé.

Les fichiers de ressources ainsi créé doivent se trouver dans un sous répertoire */lang* du module.

Les fichiers respecteront la syntaxe suivante :

- le nom du fichier reprend le nom du module (nom du fichier .dmc sans son extension)
- une première extension indique le nom de la langue du fichier de ressource (langue en Anglais, exemple : .english, .french,...)
- une seconde extension, `.lang`, permet d'identifier les fichiers de ressources.

Exemple de nom de fichier de ressources : `test.french.lang`

Chaque ligne du fichier de ressources correspond à une référence et à la traduction du message correspondant dans la langue spécifiée dans le nom du fichier.

Le kit de localisation permet aussi de gérer des messages avec des paramètres. Le téléchargement des fichiers du serveur vers les clients est automatique. Le client récupère les fichiers dans la langue de son SCOL Voy@ger. Lorsqu'il change de langue, les fichiers correspondant à la nouvelle langue sont téléchargés automatiquement.

Chez le client, les fichiers de ressources sont stockés sous la forme `nomDuModule.lang` directement dans le répertoire du module.

➢ **Syntaxe du fichier de ressources :**

Si vous voulez insérez des commentaires, commencez la ligne par `#`.

Par défaut les références sont accessibles à la fois côté client et côté serveur. Pour rendre les références accessibles uniquement côté client, commencer le nom de la référence par une étoile. (n'utilisez pas l'étoile lors de l'appel de la fonction de localisation, celle-ci est automatiquement supprimée lors du chargement des fichiers en mémoire).

Au sein d'un message, on utilise `\n` pour le retour chariot.

Les espaces en début de message, en fin de message ou deux ou plus espaces qui se suivent ne seront pas pris en compte. Il est toutefois possible d'utiliser le `\[espace]` pour y remédier mais il est préférable d'ajouter les espaces directement dans le code des programmes, ce qui évite d'éventuels oublis lors de la traduction des fichiers de ressources.

Les paramètres sont définis à l'aide de la syntaxe `<#no:texte>`

- `no` est un entier représentant le no du paramètre (qui commence à 0)
- `texte` est une chaîne de caractère permettant de préciser la nature du paramètre (cette chaîne ne doit contenir aucun espace : utiliser le _ par exemple comme séparateur)

Il est ainsi possible d'insérer un paramètre plusieurs fois. Les paramètres peuvent être insérés dans n'importe quel ordre.

➢ **API Serveur :**

`S _loc module DMI reference S parameters [S r1]]`
Permet de localiser le serveur dans la langue du SCOL Voy@ger serveur.
Retourne la localisation de la référence (2ème paramètre de la fonction) avec des paramètres insérés (3ème paramètre de la fonction).

`S _locCli module DMI client CLIENT reference S parameters [S r1]`
Permet d'envoyer au client une référence localisée dans sa langue (langue du SCOL Voy@ger du client si disponible, sinon langue du SCOL Voy@ger du serveur)
Retourne la localisation de la référence (3ème paramètre de la fonction) avec des paramètres insérés (4ème paramètre de la fonction). Si la référence n'existe pas dans la langue du client, retourne la référence traduite dans la langue du serveur.

`S _locCliEx module DMI language S reference S parameters [S r1]`
Identique `_locCli`, mais on spécifie la langue du client Par exemple, pour envoyer des email à un client dans sa langue alors qu'il n'est pas connecté, on spécifie la langue du client (2ème paramètre de la fonction) en la récupérant dans une base de données serveur. Retourne la localisation de la référence (3ème paramètre de la fonction) avec des paramètres insérés (4ème paramètre de la fonction). Si la référence n'existe pas dans la langue du client, retourne la référence traduite dans la langue du serveur.

`I _locAddRef module DMI language S référence S content S`
Permet d'ajouter dynamiquement une référence à localiser (3ème paramètre de la fonction) avec son contenu (4ème paramètre) pour une langue donnée (2ème paramètre). Retourne 1 si ajout effectué, 0 sinon.

`I _locDelRef module DMI language S reference S`
Permet de supprimer une référence (3ème paramètre de la fonction) dynamiquement pour une langue donnée (2ème paramètre). Retourne 1 si suppression effectué, 0 sinon.

`I _DMSreinitLoc module DMI`

Recharge dynamiquement les fichiers de localisation sur le serveur et chez les clients connectés.
En cas d'erreur, les fonctions renvoient :

- `"!!ERR_REF!!"` quand la référence est inexistante
- `"!!ERR_PARAM!!"` quand un paramètre est manquant

➢ **Autres services :**

`I _DMStime`
Retourne l'heure du serveur

`I _DMStickcount`
Retourne la valeur `tickcount` du serveur

`I _DMSservice client CLIENT message S`
Envoie un message de service à un client. Ce message apparaît dans une boite de dialogue.

`ObjFont Font`
Fonte principale du site.

`S DMSpath`
Chemin du fichier Dms

`S _DMSgetpath nom_fichier S`
Retourne le chemin d'un nom de fichier

`[S r1] _DMSrelativpath path S liste_fichiers [S r1]`
A partir d'un chemin par défaut et d'une liste de fichiers éventuellement relatifs (dont le nom commence par `./`), la fonction retourne une liste de noms de fichiers absolus.

`S _adderror message S`
Ajoute un message d'erreur au livre de bord. Si celui-ci est produit lors du lancement du serveur, le serveur s'arrêtera dès la fin de l'initialisation de chaque module.

`S _addwarning message S`
Ajoute un message d'avertissement au livre de bord.

`S _logBook`
Retourne le contenu du livre de bord.

➢ **Fonctions à définir dans le module :**

`IniDMI (param S)`
Fonction appelée lors du lancement de l'instance. Le paramètre qui était précédemment le nom du fichier Dmi n'est plus utilisé.

14.6.2. API client

14.6.2.1. Structures

- `DMI` : instance d'un module
- `User` : user
- `UserI` : user instance
- `RSC` : requête de téléchargement

14.6.2.2. Variables de l'API

- `this DMI` : instance courante
- `DMSname S` : nom du site
- `DMSwin ObjWin` : fenêtre de base du serveur
- `DMSlogin S` : login du client
- `DMSid I` : numéro Id du client (index constant durant toute la connexion)
- `DMSactiveX I` : vaut 1 si le client est un activeX (ou un plugin netscape)

14.6.2.3. Constantes de l'API

- `USER_global`
- `USER_client`
- `USER_changeClass`
- `USER_changeParam`
- `USER_changeAll`

14.6.2.4. Fonctions de l'API

`S _DMSgetName module DMI`
Retourne le nom court d'un module

`S _DMSgetClass module DMI`
Retourne le nom du fichier dmc d'un module

`I _DMSdelete module DMI`
Auto destruction du module client

`I _DMSsend this DMI message Com`
Envoie un message vers l'instance du même module sur un certain client

➢ **Messages inter-modules : déclenchements d'événements directs :**

`I _DMSevent this DMI event S param S reply S`
Active un événement en précisant un paramètre et une réponse. Si le paramètre vaut `nil`, c'est le paramètre par défaut qui est utilisé. Le champ `reply` devrait être laissé à `nil`.

`I _DMSeventTag module event S param S others [User r1] [callback fun[param S others [User r1]] I flag timeout]`
Provoque un événement accompagné éventuellement d'un tag. Le flag est inutilisé.

`I _DMStagKeepAlive tag Tag`
Indique que le tag doit être conservé même s'il ne lui est pas répondu immédiatement.

`I _DMStagForget tag Tag`
Indique que le tag peut-être oublié.

`I _DMSreplyTag tag Tag param S others [User r1] holdon I`
Répond au tag en lui passant un paramètre, et en indiquant si le tag doit être détruit (`holdon=0`) ou si l'on compte s'en resservir (`holdon=1`).

I _DMSdefineActions module DMI liste_actions [[S fun [module_émetteur DMI action S param S others [User r1] tag Tag] I] r1]
Définit une liste d'actions associées à des callbacks. Cette définition est incrémentale : on peut l'appeler plusieurs fois, et n'importe quand.

I _DMSremoveActions module DMI actions [S r1]
Supprime des actions de la liste.

I _DMSregister (module DMI) (cb_beforeclose fun [] I)
I _DMSregisterDMI (module DMI) (cb_action fun [from DMI action S param S reply S] I) (cb_beforeclose fun [] I)
enregistre les callbacks du module.
La fonction `beforeclose` est appelée avant que le module serveur ne soit fermé, c'est-à-dire avant la fermeture du serveur.
La callback action recevra tous les messages actions reçus par le module. La nouvelle fonction `_DMSdefineActions` rend cette callback obsolète.

- **Users :**

User UcreateUser
Crée un user virtuel local.

I UgetId user User
Retourne l'Id

I UgetFlag user User
Retourne le flag associé au user : il est une combinaison des deux constantes :

- `USER_global` : user global (par opposition à local)
- `USER_client` : user client (par opposition à virtuel)

- **UsersInstances :**

[UserI r1] Ulist module DMI
Retourne la liste des userI associés à un module

UserI UcreateUI module DMI user User class S parameters [[S r1] r1]
Crée une instance pour un User local (ceci a généralement peu d'application)

User UgetUser userI UserI
Retourne le user associé

DMI UgetLocation userI UserI
Retourne la location associée

[[S r1]r1] UgetParams userI UserI
Retourne les paramètres associés

[S r1] UgetParam userI UserI champ S
Retourne la valeur du champ

UserI UgetUserI module DMI user User
Retourne le `userInstance` associé à un module et à un user.

I Udelete userI UserI

Détruit une instance locale

I UchgClass userI UserI classe S params [[S r1] r1]
Change la classe d'un UserI local.

I UsetParams userI UserI params [[S r1] r1]
Change tous les paramètres d'une instance locale.

I UsetParam userI UserI champ S param [S r1]
Change un champ d'une instance locale.

UserI UcbDelete ui UserI callback fun [UserI] I
Définit une callback pour être informé de la destruction d'une instance. Ceci se produit généralement lorsque le serveur a appelé la fonction `Udelete`.

UserI UcbMessage ui UserI liste_de_messages [[S callback fun [ui UserI action S param S] I] r1]
Définit des callbacks sur des réceptions de messages. Ces callbacks sont simplement concaténées à la liste présente et masquent éventuellement des définitions précédentes

UserI UremoveMessage ui UserI message S
Supprime une callback sur un message.

I UsendMessage ui UserI action S param S
Envoie un message à l'instance serveur.

I UcbCreate module callback fun [ui UserI] I
Callback signalant la création d'une instance globale sur un client.

I UcbChanged ui callback fun [ui UserI flag I valeur S] I
Callback signalant un changement de classe/paramètre pour une instance globale. Le flag vaut :

- `USER_changeClass` : la classe a changé
- `USER_changeParam` : au moins un paramètre a changé (si un seul paramètre, son nom est dans le champ valeur, sinon le champ valeur vaut `nil`)
- `USER_changeAll` : vaut `USER_changeClass | USER_changeParam`

➢ **Gestion des zones :**

[ObjWin I I I I] _DMSgetZone (this DMI, zone S, conflict fun [zone S] I, resize fun [coord [win x y w h] zone S] I, destroy fun [zone S] I)
Demande une zone (le nom à passer est celui de la zone définie dans le fichier dmi)
la callback `conflict` est activée lorsque la zone est demandée par un autre module. La callback `resize` est activée lorsque la zone a changé de taille

I _DMSreleaseZone (this DMI, zone S)
Libère une zone (le nom à passer est celui de la zone définie dans le fichier dmi)

➢ **Gestion du téléchargement de ressources :**

RSC _RSCdownload module nom fichier callback fun[S] I salve
RSC _RSCdownloadP module nom fichier callback fun[S] I salve priorité I
demande le téléchargement d'une ressource en donnant le nom, le fichier dans lequel la ressource doit être stockée, et la callback lorsque le téléchargement est terminé. Le paramètre priorité indique un ordre de téléchargement : les requêtes de priorité 0 passent avant les autres.

Le paramètre `salve` ne sert plus.

Si le fichier est déjà là, avec la bonne signature, la callback peut être appelée de manière synchrone. Les fonctions `_RSCdownload` et `_RSCdownloadP` retournent :

- `nil` si la callback a été appelée,
- un objet RSC sinon : le téléchargement est en cours. Il est possible de l'interrompre.

Si le nom vaut `nil`, c'est un téléchargement de synchronisation, dont la callback sera appelée lorsque tous les téléchargements demandés précédemment (et de priorité inférieure) auront été effectués.

Si le fichier vaut `nil`, la ressource est toujours téléchargée, n'est pas stockée sur le disque et est passée à la callback, alors que sinon, c'est le nom du fichier qui est passé à la callback (`nil` si téléchargement impossible).

`I _RSCabort module ressource RSC`
Interrompt le téléchargement d'une ressource

`I _RSCabortDMI module`
Interrompt les téléchargements en cours pour le compte d'un module

`I _DMSupload module DMI nom_document S contenu S callback fun [I] I`
Transmet un document au serveur en spécifiant un nom et un contenu. La callback est appelée une fois que l'opération est terminée (l'argument vaut 1 si succès, 0 sinon). Ceci se fera via le protocole Http (requête POST)

➢ **Fonctions de localisation :**

`S _loc module DMI reference S parameters [S r1]`
Permet de localiser le client dans la langue du SCOL Voy@ger client (langue du SCOL Voy@ger serveur si la langue du client est inexistante). Retourne la localisation de la référence (2ème paramètre de la fonction) avec des paramètres insérés (3ème paramètre de la fonction).

En cas d'erreur, les fonctions renvoient :

- `"!!ERR_REF!!"` quand la référence est inexistante
- `"!!ERR_PARAM!!"` quand un paramètre est manquant

➢ **Autres services :**

`I _DMStime`
Retourne l'heure du serveur (approximation)

`I _DMStickcount`
Retourne la valeur tickcount du serveur (approximation)

`ObjFont Font`
Fonte principale du site

`ObjCursor StdCursor`
Curseur normal

`ObjCursor HandCursor`
Curseur en forme de main

`ObjCursor CrossCursor`
Curseur en forme de croix

`S _DMSgetpath nom_fichier S`
Retourne le chemin d'un nom de fichier

`[S r1] _DMSrelativpath path S liste_fichiers [S r1]`
A partir d'un chemin par défaut et d'une liste de fichiers éventuellement relatifs (dont le nom commence par `./`), la fonction retourne une liste de noms de fichiers absolus.

➢ **Fonctions à définir dans le module :**

`IniDMI (param S)`
Fonction appelée lors du lancement de l'instance. Le paramètre est celui envoyé par `_DMScreateClientDMI`.

14.6.3. API Editeur

Ecrire un éditeur de module n'est pas très compliqué : rappelons que le seul but d'un éditeur de module est de créer des blocs de définitions (au moins un bloc "dmi").

On s'appuiera sur une librairie appelée 'templateEdit1' qui offre une Api simple et une interface graphique propre et homogène :

- il faudra définir la fonction `IniEditor` qui sera la fonction d'initialisation de l'éditeur
- cette fonction appellera dès le début la fonction `startEditor` avec les paramètres appropriés, et notamment la définition de deux callbacks:
  - `load` : callback appelée lors du chargement du module. Cette fonction a accès aux valeurs actuelles des blocs de définition du module
  - `save` : callback appelée lors de la sauvegarde et qui doit définir les nouveaux blocs de définition.
- la fonction `IniEditor` initialisera ensuite les interfaces graphiques spécifiques au module
- enfin elle se terminera par un appel à la fonction `openDMI` qui se chargera d'appeler la callback `load`.

Cette API est détaillée ici :

14.6.3.1. Fonctions appelables

```
Editor startEditor
  Chn channel
  ObjWin parent
  I x                    /* main window left bound */
  I y                    /* main window upper bound */
  I w                    /* edit window width */
  I h                    /* edit window height */
  I winflag              /* main window flags (see the ObjWin creation)*/
  I flag                 /* flags for statusbar, ... */
  S ___                  /* unused */
  S _class               /* class file, only used if compatibility with older SCS versions (<2) is required*/
  S _help                /* help file , only used if compatibility with older SCS versions (<2) is required*/
  S _icone          /* default icon file , only used if compatibility with older SCS versions (<2) is required*/
  fun [[[S r1] r1]] I load     /* load callback */
  fun [S S] [[S r1] r1] save   /* save callback */
  [
  [[[I S fun [ObjMenuItem Editor] I] r1] [[I S fun [ObjMenuItem Editor] I] r1]]
```

[[[I S fun [ObjMenuItem Editor] I] r1] [[I S fun [ObjMenuItem Editor] I] r1]]
[[S [ObjMenu r1]] r1]
] *additional menus* /* additional menus (only used in older versions (<2))*/

Lance le "*template editor*" avec en paramètres principaux la taille de la zone d'édition propre au module, et les deux callbacks `load` et `save` :

- callback `load` : reçoit en paramètre le bloc de définition "Dmi"
- callback `save` : doit retourner le bloc de définition "Dmi"

La fonction retourne un objet de type **Editor**.

`ObjWin getEditWin editor Editor`
Renvoie la fenêtre fille de l'éditeur dans laquelle l'interface propre à la classe peut être affichée. A partir de cet objet on peut facilement obtenir la taille client et définir une callback pour le redimensionnement de l'éditeur.

`I setEditorStatus editor Editor message S`
permet d'afficher des messages dans la barre d'état de l'éditeur. Ceci suppose que le flag correspondant a été positionné lors de l'appel à `startEditor`.

`I openDMI  editor Editor`
Charge le module, en appelant notamment la callback `load`.

`[[S r1] r1] getDef editor Editor name_of_block S`
Cette fonction sera typiquement appelée dans la callback `load`, pour lire les blocs de définitions autres que le bloc "Dmi"

`I setDef editor Editor name_of_block S content [[S r1] r1]`
Cette fonction sera typiquement appelée dans la callback `save` pour définir des blocs de définitions autres que le bloc "Dmi".

`S _locEditor reference S parameters [S r1]`
permet de localiser l'éditeur dans la langue du SCOL Voy@ger. Retourne la localisation de la référence (2ème paramètre de la fonction) avec des paramètres insérés (3ème paramètre de la fonction).

14.6.3.2. Fonctions à définir

`IniEditor(param S)`
Fonction appelée lors du lancement de l'éditeur. Le paramètre n'est plus utilisé.

14.7. Exemple 1 : module fonctionnant uniquement sur le serveur.

L'exemple suivant est un module qui gère un automate capable de recevoir des messages, de les analyser et d'y répondre. Typiquement, on utilise cet automate en le connectant sur une cellule de chat : l'automate "entend" ce qui se dit et peut répondre soit à un seul utilisateur, soit à tous.

Fichier : ***Dms/Tutorial/Bot0/bot0.dmc***

```
name Bot
serverNeeded ./bot0.pkg
serverLoad ./bot0.pkg
editorLoad _load\ "Dms/L/templateEdit1.pkg"\n_load\
"locked/lib/enterbox.pkg"\n_load\ "./bot0edit.pkg"
bitmap ./bot0.scs.bmp
tree ./bot0.scs.bmp
helpFile ./help.txt
```

```
version 2 2
```

Fichier : ***Dms/Tutorial/Bot0/bot0.pkg***

```
/* Bot0 - DMS - May 98 - by Sylvain HUET */
/* Mar 00 update */

/* >>>>> PLEASE DO NOT CHANGE THE FOLLOWING FONCTIONS */
fun broad(text)=_DMSeventTag this nil "broad" text nil nil;;
fun private(u,text)=_DMSeventTag this u "private" text nil nil;;
fun user(event,u,text)=_DMSeventTag this u event text nil nil;;

/* >>>>> PLEASE INSERT ONLY HERE YOUR OWN CODE */

/* this function is called when someone says something :
 -u is the visitor who has spoken
 -text is the text the visitor said
*/

fun hear_bot(from,u,action,text,ulist,tag)=
if !strcmp text "foo" then
 (private u "<Bot> bar\n";
  0)
 else if !strcmp text "time" then
 (broad strcat "<Bot> " ctime time;
  0)
 else let hd strextr text -> l in
 if !strcmp hd l "square" then
 (let atoi hd tl l -> x in broad strcat "<Bot> " itoa x*x;
  0)
 else nil;;

/* this function is called when someone is in :
 -u is the visitor who got in
*/
fun in_bot(from,u,action,param,ulist,tag)=
 broad strcat "<Bot> Hello " _DMSgetLogin UtoC u;
 private u "<Bot> You can ask me for the time or to calculate integer
squares : type 'time' or 'square 4'\n";;

/* this function is called when someone is out :
-u is the visitor who got out
*/
fun out_bot(from,u,action,param,ulist,tag)=
 broad strcat "<Bot> Good bye " _DMSgetLogin UtoC u;;


/* >>>>> PLEASE DO NOT CHANGE THE FOLLOWING CODE */

fun IniDMI(file)=
 _DMSregister this nil nil nil;
 _DMSdefineActions this ["in" @in_bot]::["out" @out_bot]::["hear"
@hear_bot]::nil;;
```

Fichier : *Dms/Tutorial/Bot0/bot0edit.pkg*

```
/* Bot Editor - DMS - May 98 - by Sylvain HUET */
/* Rev. Aug. '98 - by Marc BARILLEY */
/* Rev. Sep. '00 - by Julien ZORKO */
```

```
typeof bannerll=ObjText;;
typeof ll=ObjList;;
typeof addll= ObjButton;;
typeof delll = ObjButton;;
typeof links=[S r1];;

fun updatel2(s,b)=
 _ADDlist ll 1000 s;;

fun updatel()=
 _RSTlist ll;
 apply_on_list links @updatel2 0;;

fun addr(s)=
 if s==nil then nil
 else
 (set links=conc links s::nil;
  updatel);;

fun _add(but, editWin)=
 iniEnterBox _channel editWin nil nil "New Event" @addr "Enter a new event
name" ;;

fun _rem(x,b)=
 let _GETlist ll ->[i _] in
 let nth_list links i -> a in
 if a==nil then nil else
 (set links=remove_from_list links a;
  updatel);;

fun fdlink(a,b)=
 if strcmp hd a "botevent" then 0
 else (set links=conc links (hd tl a)::nil; 0);;

fun load (l) =
  set links=nil;
  apply_on_list l @fdlink 0;
  updatel;
  0;;

fun suppevent(l)=
  if l==nil then nil
  else let l->[a n] in ("botevent"::a::nil)::("event"::a::nil)::suppevent
n;;

fun save (filename, n)=
  ("action"::"in"::nil)::
  ("action"::"out"::nil)::
  ("action"::"hear"::nil)::
  ("event"::"broad"::nil)::
  ("event"::"private"::nil)::
  suppevent links;;

fun rflSizeEditWin (wn, blurp, w, h)=
  _SIZEtext bannerll w-10 20 5 5;
  _SIZElist ll w-10 h-50 5 25;
  _SIZEbutton addll 45 20 5 h-25;
  _SIZEbutton delll 45 20 70 h-25;;

fun IniEditor(s)=
  let [315 340] -> [w h] in
```

```
  let startEditor
    _channel nil nil nil 315 340 WN_NORMAL-WN_SIZEBOX EDITOR_NORMAL
    s nil nil nil
    @load @save nil
    -> ed in
  let getEditWin ed -> editWin in
  (
    _CBwinSize editWin @rflSizeEditWin 0;

    set bannerll = _CRtext _channel editWin 5 5 w-10 20
                           ET_ALIGN_CENTER "New events :";
    set ll = _CRlist _channel editWin 5 25 w-10 h-50
                     LB_DOWN+LB_VSCROLL;
    set addll = _CBbutton _CRbutton _channel editWin
                                    5 h-25 45 20 0 "Add" @_add editWin;
    set delll = _CBbutton _CRbutton _channel editWin
                                    70 h-25 45 20 0 "Remove" @_rem 0;
    if s==nil then nil else openDMI ed
  );
0;;
```

Cet exemple est volontairement très simple.

Le fichier Dmc ne comporte pas de surprise, il reprend point pour point ce qui a été expliqué lors de la description du format Dmc. Contentez vous simplement de relever quels fichiers sont utilisés par le module et comment sont définis les scripts.

Le fichier bot0.pkg est le cœur du programme : c'est celui qui fonctionne sur le serveur. La fonction `IniDmi` définit simplement les callbacks sur les trois actions que le module reconnaît : `in`, `out` et `hear` :

- `in` : cette entrée indique au robot que quelqu'un vient d'arriver
- `out` : cette entrée indique au robot que quelqu'un vient de partir
- `hear` : cette entrée indique au robot qu'un message vient d'être "entendu".

Le module appelle en conséquence trois fonctions, `in_bot`, `out_bot` et `hear_bot`. L'idée est la suivante : si le module bot est connecté correctement, la fonction `in_bot` sera appelée chaque fois qu'un visiteur entrera dans le monde, la fonction `out_bot` sera appelée chaque fois qu'un visiteur sortira du monde. La fonction `hear_bot` sera appelée chaque fois qu'un visiteur dira quelque chose.

Ces trois fonctions ont un argument `u`, de type User. Pour récupérer le nom d'un utilisateur, on convertira le User en CLIENT avec la fonction `CtoU` et on utilisera la fonction `_DMSgetLogin` qui prend en argument un type CLIENT et retourne un type chaîne de caractères (voir API serveur)

Dans la fonction `hear_bot`, le texte entendu est passé dans l'argument 'text'. Pour découper cet argument en mots, en vue d'une analyse syntaxique, il suffit de lui appliquer les deux fonctions '`hd strextr`'. Le résultat est une liste de chaîne de caractères (type `[S r1]`). Pour analyser cette liste, on utilise les classiques fonctions `hd` et `tl`. On peut utiliser également la fonction `nth_list` (voir librairie standard).

Le module définit trois fonctions :

- `broad` : envoie un message à tous
- `private` : envoie un message à un seul
- `user` : déclenche un événement concernant un utilisateur.

Vous pouvez vous entraîner en modifiant le code des fonctions `in_bot`, `out_bot` et `hear_bot`.

Le fichier bot0edit.pkg est montre comment définir un éditeur à partir de fonctions prédéfinies dans le fichier '`Dms/L/templateEdit1.pkg`'. Dans cet exemple déjà complexe d'éditeur, il s'agit d'offrir à l'utilisateur la possibilité de créer de nouveaux événements. La fonction importante est `startEditor`. Cette fonction crée la fenêtre standard de l'éditeur (accessible par la fonction `getEditWin )`, avec boutons ok, appliquer, annuler et aide, icône, barre d'état ... On lui donne entre autre argument deux callback `load` et `save`.

La callback `load` est appelée au chargement du fichier Dmi : elle prend en argument le contenu du fichier Dmi auquel on a déjà appliqué la fonction `strextr`. Le type de l'argument est donc `[[S r1] r1]`.

La callback `save` est appelée lors de la sauvegarde du fichier Dmi. Elle doit retourner une suite de lignes décrivant, les événements, les actions, les zones, ainsi que les paramètres spécifiques du module. Ce résultat doit être de type `[[S r1] r1]`.

```
startEditor
    channel nil 0 0 315 340 WN_SIZEBOX EDITOR_NORMAL
    s "Dms/Bots/Bot0/bot0.dmc" "Dms/Bots/Bot0/help.txt"
      "Dms/Bots/Bot0/bot0.bmp"
    @load @save nil;
```

Le reste du fichier bot0edit.pkg gère un objet graphique de type liste, ainsi que 2 boutons. Pour cela, on peut utiliser les fonctions de l'API2D sur l'ObjWin retourné par la fonction `getEditWin`, ce qui permet de récupérer les dimensions de la zone éditable et de définir une callback de `resize` qui redimensionnera et repositionnera les éléments graphiques lorsque la fenêtre de l'éditeur sera redimensionnée par l'utilisateur.

Exercices :

- **quicksort** : demander au robot de trier une liste de mots
- **calculatrice** : plutôt que de calculer un carré, prendre une expression arithmétique (en polonaise inversée pour simplifier)
- **guide** : demander au bot de vous téléporter quelque part
- **bonnes mœurs** : téléporter automatiquement vers un cachot un visiteur qui a dit 'merde'
- **passer avec succès le test de Turing**.

14.8. Exemple 2 : module distribué et gestion de zones

L'exemple suivant est un module distribué. Il définit un bouton sur l'interface du client et/ou du serveur.

Fichier : *Dms/Tutorial/Button/button.dmc*

```
name Button
register ./buttonc.pkg
serverNeeded
serverLoad ./buttons.pkg
clientNeeded
clientLoad ./buttonc.pkg
editorLoad _load\ "Dms/L/templateEdit1.pkg"\n_load\ "./buttonedit.pkg"
bitmap ./button.scs.bmp
tree ./button.scs.bmp
helpFile ./help.txt
version 2 2
```

Fichier : *Dms/Tutorial/Button/buttons.pkg*

```
/* Button Server - DMS - march 98 - by Sylvain HUET */
/* Rev. 0101 - Aug. '98 - by Marc BARILLEY */

fun start(from,u,action,param,ulist,tag)=
  if _DMScreateClientDMI this UtoC u nil then
   _DMSeventTag this u "started" nil nil nil
  else nil;;

fun end(from,u,action,param,ulist,tag)=
  if _DMSdelClientDMI this UtoC u then
   _DMSeventTag this u "ended" nil nil nil
  else nil;;

fun IniDMI(file)=
 _DMSregister this nil nil nil;
 _DMSdefineActions this ["start" @start]::["end" @end]::nil;;
```

Fichier : *Dms/Tutorial/Button/buttonc.pkg*

```
/* Button Client - DMS - March 98 - by Sylvain HUET */
/* Rev. 0101 - Aug. '98 - by Marc BARILLEY */

typeof button=ObjButton;;

fun pressbut(a,b)= _DMSeventTag this "click" nil nil nil;;

fun _end(s)=
  _DMSdelete this;;

fun _resizeI(x,s)=
 let x->[win x y w h] in _SIZEbutton button w h x y;
 0;;

fun IniDMI(param)=
 let _DMSgetZone this "Button" @_end @_resizeI @_end ->[win x y w h] in
 if win==nil then nil else
  set button=_CBbutton _CRbutton _channel win x y w h 0 _DMSgetName this
@pressbut 0
 ;;
```

Fichier : *Dms/Tutorial/Button/buttonedit.pkg*

```
/* Button Editor - DMS - Mar 98 - by Sylvain HUET */
/* Rev. Aug. '98 - by Marc BARILLEY */

fun save (filename, n)=
  ("action"::"start"::nil)::
  ("action"::"end"::nil)::
  ("eventC"::"click"::nil)::
  ("event"::"started"::nil)::
  ("event"::"ended"::nil)::
  ("zoneC"::"Button"::nil)::
  nil;;

fun IniEditor(s)=
```

```
  let startEditor
    _channel nil nil nil 315 0 WN_NORMAL-WN_SIZEBOX EDITOR_NORMAL
    s nil nil nil
    nil @save nil
    -> ed in
  if s==nil then nil else openDMI ed;
0;;
```

Dans le fichier buttons.pkg, on remarque deux éléments importants :

- l'utilisation des zones dans la fonction IniDMI : `_DMSgetZone this "Button" @_end @_resizeI @_end`
- la création et la destruction du module client dans les fonctions `start` et `end` du serveur.

On remarque aussi la forme minimale de l'éditeur : en effet, l'éditeur n'a pas besoin d'interface utilisateur spécifique.

14.9. Exemple 3 : module distribué et message intra-module

L'exemple suivant est un module capable de poser une question à l'utilisateur, sous forme d'une boite de message. Nous nous intéressons particulièrement à l'échange de messages entre le module client et le module serveur.

Fichier : *Dms/Tutorial/Quizz/quizz.dmc*

```
name Quizz
register ./quizzc.pkg
serverNeeded
serverLoad ./quizzs.pkg
clientNeeded
clientLoad ./quizzc.pkg
editorLoad _load\ "Dms/L/templateEdit1.pkg"\n_load\ "./quizzedit.pkg"
bitmap ./quizz.scs.bmp
tree ./quizz.scs.bmp
helpFile ./help.txt
version 2 2
```

Fichier : *Dms/Tutorial/Quizz/quizzs.pkg*

```
/* Quizz Server - DMS - march 98 - by Sylvain HUET */

defcom Cquizz=quizz S I;;
struct Qz=[cliQz:CLIENT,txtQz:S,numQz:I]mkQz;;

typeof quizz=[S r1];;
typeof qu=[Qz r1];;

fun byboth(a,z)=let z->[c i] in c==a.cliQz && i==a.numQz;;
fun __answer(i,yes)=
 let search_in_list qu @byboth [DMSsender i] -> x in
 if x==nil then nil
 else
 (set qu=remove_from_list qu x;
  _DMSevent this DMSsender strcat if yes then "yes" else "no" itoa i nil
nil);;

fun removecli(l,c)=
 if l==nil then nil else let l->[a n] in
```

```
 if a.cliQz==c then removecli n c else a::removecli n c;;

fun logout(cli)=
 set qu=removecli qu cli;
 0;;

fun in(from,u,action,param,ulist,tag,i)=
 let UtoC u-> cli in
 let if i==nil then param else nth_list quizz i -> txt in
 (_DMScreateClientDMI this cli nil;
  _DMSsend this cli Cquizz [txt i];
  set qu=(mkQz[cli txt i])::qu;
  0);;

fun getQuizz(l,i)=
 if l==nil then nil
 else let l->[q n] in
 if !strcmp hd q "quizz" then
 (_DMSdefineActions this [strcat "in" itoa i mkfun7 @in i]::nil;
 (hd tl q)::getQuizz n i+1)
 else getQuizz n i;;

fun IniDMI(file)=
 let _DMSgetDef this "dmi" ->l in
  (set quizz=getQuizz l 0);
 _DMSregister this nil @logout nil;
 _DMSdefineActions this ["in" mkfun7 @in nil]::nil;;
```

Fichier : *Dms/Tutorial/Quizz/quizzc.pkg*

```
/* Quizz Client - DMS - March 97 - by Sylvain HUET */

defcom Canswer=answer I I;;

fun IniDMI(param)=0;;

fun res(x,i,r)=
 _DMSsend this Canswer [i r];;

fun __quizz(s,i)=
  _DLGrflmessage _DLGMessageBox _channel DMSwin "Question" s 2 @res i;;
```

Fichier : *Dms/Tutorial/Quizz/quizzedit.pkg*

```
/* Quizz Editor - DMS - feb 98 - by Sylvain HUET */

typeof editWin=ObjWin;;
typeof quizz=tab ObjText;;

fun onequizz(i,x)=
 _CRtext _channel editWin 5 25+i*25 10 20 ET_ALIGN_CENTER itoa i;
 _CReditLine _channel editWin 20 25+i*25 290 20 ET_DOWN+ET_AHSCROLL "";;

fun createQuizz()=
 set quizz=create_tab 8 @onequizz 0;;

fun getQuizz(l)=
```

```
 if l==nil then nil
 else let l->[q n] in
 if !strcmp hd q "quizz" then (hd tl q)::getQuizz n
 else getQuizz n;;

fun setQuizz(l,i)=
 if l==nil || i>=8 then 0
 else let l->[a n] in
 (_SETtext quizz.i a;
  setQuizz n i+1);;

/* SCS editor */
fun load (l) =
  setQuizz getQuizz l 0;
  0;;

fun getText(i)=
 if i==8 then nil
 else
 ("action"::(strcat "in" itoa i)::nil)::
 ("event"::(strcat "yes" itoa i)::nil)::
 ("event"::(strcat "no" itoa i)::nil)::
 ("quizz"::(_GETtext quizz.i)::nil)::getText i+1;;

fun save (filename, n)=
  ("action"::"in"::nil)::
  ("event"::"yes"::nil)::
  ("event"::"no"::nil)::
  getText 0;;

fun IniEditor(s)=
  let startEditor
    _channel nil nil nil 315 230 WN_NORMAL-WN_SIZEBOX EDITOR_NORMAL
    s nil nil nil
    @load @save nil
    -> ed in
  (
    set editWin = getEditWin ed;
    _CRtext _channel editWin 20 5 290 20 ET_ALIGN_CENTER "Prompt";
    createQuizz;
    if s==nil then nil else openDMI ed
  );
0;;
```

Dans cet exemple, le module serveur envoie au module client un message contenant le texte de la question ainsi qu'un identifiant de question :

```
_DMSsend this cli Cquizz [txt i];
```

Le message est créé à partir du constructeur de communication Cquizz, défini par :

```
defcom Cquizz=quizz S I;;
```

Lorsque le module client le reçoit, il exécute la fonction `__quizz`. Celle-ci ouvre une boite de dialogue avec le texte de la question. Lorsque l'utilisateur répond, la fonction `res` est appelée et transmet au serveur la réponse :

```
_DMSsend this Canswer [i r];;
```

Le serveur exécute alors la fonction `__answer`, dans laquelle la variable `DMSsender` de type `CLIENT` contient le client émetteur du message.

14.10. Module C3d3 et plugins

14.10.1. Concept de l'API 3D

Le module C3d est un module très important : il est capable de gérer un espace 3d contenant des avatars et des objets animés, c'est donc le module le plus visible.

Ce module gère évidemment une scène 3d, mais il offre un système de plugins qui permet à un développeur d'interfacer facilement de nouvelles fonctionnalités dans l'espace 3d. Ce mécanisme repose sur le système de Users et de UserInstances décrit précédemment :

- chaque fonctionnalité du site sera considérée comme un User :
  - pour un objet qui tourne, il faut imaginer qu'il y a dans la scène un User virtuel dont le seul rôle est de faire tourner l'objet. Ce User est paramétré par :
    - l'objet à faire tourner
    - les valeurs de cette rotation (axe, vitesse, ...)
- on trouve donc un User client par avatar, et un User virtuel par fonctionnalité
- pour utiliser chacun de ces Users, le module C3d3 définit un UserInstance par User. Un UserInstance est défini par :
  - un User
  - un nom
  - une classe
  - des paramètres divers
  - une visibilité

On utilisera la classe pour définir la fonctionnalité : Avatar, rotation, ...

Le module C3d3 gère donc une liste de UserInstance de classes différentes. A chaque classe correspond un traitement différent, et il doit être possible pour un développeur d'ajouter facilement une nouvelle classe, avec un nouveau traitement. Pour cela, on introduit la notion de plugin C3d3 : un plugin C3d3 est un petit programme (avec une partie serveur et/ou une partie client) qui gère le fonctionnement d'une classe.

Dans la suite, on définit la structure **Ob**, qui est une surclasse des UserInstances, et on explique comment développer un plugin.

14.10.1.1. Structure Ob

La structure fondamentale du module C3d3 est la structure **Ob**. Elle représente un "objet" au sens de la programmation objet : c'est à dire une instance d'une certaine classe.

Ainsi, la structure Ob permet de décrire :

- les avatars
- des fonctionnalités diverses

En fait, la structure Ob est une surclasse de la structure UserI, et bénéficie donc du mécanisme des Users. Typiquement, les Ob avatars seront des UserInstance de User clients, alors que les Ob fonctionnalités seront des UserInstance de User virtuels.

Lorsqu'un client entre dans le module, un UserInstance est créé avec :

- pour classe, la valeur d'attente de la classe (qui aura été définie précédemment par la fonction `ObSetClass`)
- pour nom, le login du client

Au démarrage du serveur, les instances déjà présentes (définies sous l'éditeur) sont créées avec:

- la classe indiquée dans l'éditeur
- le nom indiqué dans l'éditeur
- les paramètres définis dans l'éditeur avec deux lignes supplémentaires : `name` et `anchor` (tels que décrits dans l'éditeur)

Le C3d3 intègre la gestion de visibilité du système de Users.

Les instances autres que les avatars sont toujours créées à la racine de l'arbre avec le flag de commutativité à 1.

Le module C3d3 détermine les droits de l'avatar de la manière suivante :

- on définit sous l'éditeur C3d3 (menu '*advanced*') la variable de ressources gérant les droits.
- Lorsque l'avatar est créé, on lit cette variable de ressource, qui doit avoir le format suivant :

```
Format strbuild : ((itoa commut) : :rights : :nil) : :nil
```

Où le paramètre `rights` est un chemin de type : `a.b.c` (le point est le séparateur). La chaîne vide correspond au sommet de l'arbre. (a.b.c est le fils de a.b).

Pour les tests, il y a un cas particulier : si on définit dans l'éditeur C3d3 la variable de ressources gérant les droits comme 'altern', les avatars seront successivement placés dans le chemin " 0 " et le chemin " 1 ".

Quelques paramètres définissent l'objet :

➢ **Sur le serveur :**

- le UserInstance (et donc la classe, les paramètres, des callbacks de communication avec le module UserClass et avec les UserInstance clients)
- le nom
- une callback de destruction
- la position courante (x,y,z)(a,b,c) de l'objet, lorsque cela a un sens

`UserI ObUi(Ob)`
Retourne le UserInstance associé à un objet

`I ObMobile(ob Ob)`
Retourne le flag de mobilité de l'objet

`[I I I] ObPos(ob Ob)`
Retourne la position connue de l'objet

`[I I I] ObAng(ob Ob)`
Retourne l'orientation connue de l'objet

`S ObName(ob Ob)`
Retourne le nom de l'objet

`fun [Ob] I ObCbDestroy(Ob,cb fun [Ob] I)`
Définit une callback à appeler avant de détruire l'objet

`[Ob r1] ObList`
Retourne la liste des objets.

`fun [Ob S] I ObCbSpeak callback fun [Ob S] I`
Redéfinit la fonction appelée sur le serveur sur la réception d'un message de chat

[S r1] Obgetglobalress (ress S)
Retourne la valeur d'une variable de ressource du C3d.

I fun ObSetClass user User class S
Si le user n'est pas déjà dans la cellule 3d, indique la classe du UserInstance qu'il faudra créer lorsque le user se présentera.
Si le user est déjà dans la cellule 3d, change la classe du UserInstance associé.

I ObUpdateClass user User class S
Si le user est déjà dans la cellule 3d, change la classe du UserInstance associé.

Ob ObAddInstance class S param [[S r1] r1]
Crée un nouveau User virtuel et une nouvelle instance. Parmi les paramètres, on utilisera 'name' pour déterminer le nom de l'instance et 'anchor' pour l'ancre.

I ObRemoveInstance id I
Détruit une instance en fonction du numéro id du User.

Il peut-être utile qu'un plugin (côté serveur), ayant reçu une action quelconque, décide de « faire entrer » un User dans l'espace 3d dans une position donnée (définie par un nom dans l'éditeur du module C3d3). Deux cas se présentent :

- le User est déjà dans la cellule 3d : il faut juste le déplacer à sa nouvelle position
- le User n'est pas déjà dans la cellule 3d : il faut le faire venir.

Une nouvelle fonction est définie pour cela :

I ObPlaceAvatar user User position S
On passe en paramètre le User ainsi que le nom de la position.
La valeur de retour est sans intérêt.

➢ **Sur le client :**

- le UserInstance (et donc la classe, les paramètres, des callbacks de communication avec le module UserClass et avec le UserInstance serveur)
- le nom
- un flag de mobilité : doit-on rafraîchir et synchroniser la position de l'objet ?
- un flag avatar : l'objet apparaît-il dans la liste des avatars présents dans la scène ?
- une ancre
- diverses callbacks
- un objet 3d principal, facultatif

UserI ObUi(ob Ob)
retourne le UserInstance associé à un objet

S ObName(ob Ob)
retourne le nom de l'objet

I ObAvatar(ob Ob)
retourne le flag avatar de l'objet

I ObMobile(ob Ob)
retourne le flag de mobilité de l'objet

[Anchor r1] ObAnchor(ob Ob)
retourne l'ancre associée à un objet

H3d ObSetMain ob Ob objet3d H3d
définit l'objet 3d principal

H3d ObGetMain(ob Ob)
retourne l'objet 3d principal

I ObSelect0(id I)
I ObSelect1(id I)
I ObSelect2(id)
I ObSelect3(id I)
Ces fonctions déclenchent un événement client 'selectn' avec en paramètre le numéro Id (format itoa)

I ObSendLocal from Ob to Ob action S param S rep S
Envoie un message à une instance locale.

I ObHear string S
sort un message par l'événement "hear"

I ObSetCam ob Ob
définit l'objet auquel est liée la caméra : cette fonction lie la caméra à l'objet 3d principal

Surface ObBuffer
retourne le buffer de rendu

I ObSetBackground col I
définit la couleur 24 bits de fond de rendu (`nil` : aucune).

[Ob r1] fun ObList
retourne la liste des objets.

Les fonctions suivantes permettent de définir les callbacks : il ne faut pas modifier soi-même la structure Ob :

fun ObCbGetName(o,f) : fun[Ob] S
fonction retournant le nom de l'objet

fun ObCbGetVal(o,f) : fun[Ob S] S
fonction retournant une certaine valeur

fun ObCbSetpos(o,f) : fun[Ob [I I I][I I I]] I
fonction positionnant l'objet dans une certaine position

fun ObCbAnim(o,f) : fun[Ob] I
fonction appelée avant chaque rendu

fun ObCbSend(o,f) : fun[Ob S S S]
fonction appelée lorsqu'un message est reçu

fun ObCbClick(o,f) : fun[Ob H3d HMat3d I] I
fonction appelée lorsque l'utilisateur clique sur l'objet principal ou l'un de ses descendants (handler, matériau et bouton)

`fun ObCbDclick(o,f) : fun[Ob H3d HMat3d I] I`
fonction appelée lorsque l'utilisateur double-clique sur l'objet principal ou l'un de ses descendants (handler, matériau et bouton)

`fun ObCbMove(o,f) : fun[Ob H3d HMat3d] I`
fonction appelée lorsque l'utilisateur passe la souris sur l'objet principal ou l'un de ses descendants (handler, matériau)

`fun ObCbDraw(o,f) : fun[Ob ObjSurface I I I] I`
fonction traçant l'objet en 2d sur un bitmap, sur une position et une taille données

`fun ObCbControl(o,f) : fun[Ob [[I I I] [I I I]]] I`
fonction demandant le déplacement de l'objet en passant les 2 vecteurs vitesse

`fun ObCbControlClick(o,f) : fun[Ob [H3d HMat3d I]] I`
fonction appelée chaque fois que l'utilisateur clique dans la 3d

`fun ObCbControlMove(o,f) : fun[Ob [Ob H3d HMat3d]] I`
fonction appelée chaque fois que l'utilisateur passe la souris sur un objet 3d principal. La callback retourne deux objets Ob : l'objet qui a défini la callback, puis l'objet pointé par la souris.

`fun ObCbControlKeyDown(o,f) : fun[Ob [I I]] I`
fonction appelée chaque fois que l'utilisateur appuie sur une touche dans la 3d

`fun ObCbControlKeyUp(o,f) : fun[Ob I] I`
fonction appelée chaque fois que l'utilisateur relève une touche dans la 3d

`fun ObCbSpeak(o,f) : fun[Ob S] I`
fonction appelée lorsque l'utilisateur dit quelque chose

`fun ObCbPostRender(o,f) : fun[Ob [ObjBitmap [I I]]] I`
fonction appelée après chaque rendu. Le bitmap vaut `ObBuffer`()

`fun ObCbReceiveLocal(o,f) : fun[Ob Ob S S S] I`
fonction appelée lors de communication locale entre instance (fonction `ObSendLocal`)
les paramètres de la callback sont : `from`, `to`, `action`, `param`, `rep`

`fun ObCbDestroy(o,f) : fun[Ob] I`
fonction appelée avant la destruction de l'objet

Les fonctions suivantes retourne la valeur des callbacks.

```
fun ObGetName(o) : fun[Ob] S
fun ObGetVal(o) : fun[Ob S] S
fun ObSetpos(o) : fun[Ob [I I I][I I I]] I
fun ObAnim(o) : fun[Ob] I
fun ObSend(o) : fun[Ob S S S]
fun ObClick(o) : fun[Ob H3d HMat3d I] I
fun ObDclick(o) : fun[Ob H3d HMat3d I] I
fun ObMove(o) : fun[Ob H3d HMat3d] I
fun ObDraw(o) : fun[Ob ObjBitmap I I I] I
fun ObControl(o) : fun[Ob [[I I I] [I I I]]] I
fun ObControlClick(o) : fun[Ob [H3d HMat3d I]] I
fun ObControlMove(o) : fun[Ob [Ob H3d HMat3d]] I
fun ObControlKeyDown(o) : fun[Ob [I I]] I
fun ObControlKeyUp(o) : fun[Ob I] I
```

```
fun ObSpeak(o) : fun[Ob S] I
fun ObPostRender(o) : fun[Ob [ObjBitmap [I I]]] I
fun ObReceiveLocal(o) : fun [Ob Ob S S S] I
fun ObDestroy(o) : fun[Ob] I
```

```
fun Obgetglobalress(ress S) [S r1]
```
retourne la valeur d'une variable de ressource du C3d.

Il est possible de définir des objets cliquables (c'est-à-dire des parties de la scène 3d sur lesquelles le curseur de la souris sera modifié et auxquelles sont associées des callbacks de move, click et double-click) :
```
I ObSetLinks [ob Ob liste_liens [[H3d Hmat3d S ObjCursor fun [Ob H3d Hmat3d I] I fun [Ob H3d Hmat3d I] I fun [Ob H3d Hmat3d] I] r1]
```

Chaque lien est un tuple contenant :

- handler 3d du lien
- éventuellement handler matériau du lien (si `nil`, l'objet entier est un lien, indépendamment du matériau)
- nom apparent du lien
- curseur souris à utiliser (deux constantes : `HandCursor` (une main) et `StdCursor` (la flèche simple) sont utilisables)
- callback `click` (arguments : instance, handler 3d, handler matériau, état boutons)
- callback `double-click` (arguments : instance, handler 3d, handler matériau, état boutons)
- callback `move` (arguments : instance, handler 3d, handler matériau)

```
fun ObGetLinks ob Ob
```
retourne la liste précédente

On trouve quelques variables globales :

```
session : S3d
```
session 3d

```
shell : H3d
```
nœud principal de la scène

```
cam : H3d
```
caméra

```
name3d : S
```
nom de la cellule

Lorsqu'un objet est créé sur le client, deux cas se présentent :

- **l'objet correspond à un avatar**

  S'il s'agit d'un avatar autre que celui de l'utilisateur de la machine :
  - La callback `clickStd` est définie d'office :
  - appel de `ObSelect1` sur bouton gauche, `ObSelect2` sur bouton droit.

  La callback de définition de position `setPosStd` est définie d'office. Elle suppose que l'avatar a la structure suivante :
  - un noeud shell représentant la position de l'avatar (situé normalement à hauteur de caméra)
  - un objet 3d fils oscillant autour de cette position.

  S'il s'agit de l'avatar de l'utilisateur de la machine :

La callback `controlStd` est définie d'office : gestion du déplacement avec collisions

La callback de définition de position `setPos Std` est définie d'office. Ces deux fonctions supposent que l'avatar a la structure suivante :

- un nœud shell placé aux pieds, qui tourne selon l'axe vertical
- une sphère de collision d'un mètre de rayon située à 1m10 au-dessus des pieds
- un nœud shell auquel la caméra sera attachée situé à 1m60 au-dessus des pieds, et qui tourne selon l'axe horizontal

La callback `speakStd` est définie d'office

De plus, si la classe de l'avatar n'est pas présente, la classe 'default' est utilisée : l'intérêt est de faire apparaître au plus tôt l'avatar dans la scène, même si ce n'est pas sous sa forme définitive.

Cet avatar par défaut (plaque avec logo SCOL flottant dans l'air) définit les callbacks suivantes :

- callback d'animation `CbAnim`
- callback de destruction `CbDestroy`

Lorsque la classe sera présente, l'avatar par défaut sera détruit et remplacé par l'avatar normal.

Corollaire, il est possible de surcharger l'avatar par défaut en introduisant un plugin.

- **l'objet correspond à une fonctionnalité** :

  aucune callback n'est prédéfinie

14.10.2. Ancres

En développant des fonctionnalités 3d, on remarque rapidement qu'il y a deux types de fonctions :

- celles qui sont liées spécifiquement à un objet, et qui utiliseront le champ 'objet principal', comme par exemple les avatars
- celles qui ont besoin de plus d'éléments : plusieurs objets3d/matériaux/positions, comme par exemple un module faisant suivre de manière synchrone plusieurs trajectoires à plusieurs objets 3d

Dans le deuxième type, il n'y a plus de notion d'objet principal. Celle-ci est remplacée par la notion d'ancre. Une ancre est une liste (ordonnée) d'objets, de matériaux et de positions. Pratiquement c'est une liste d'éléments de type Anchor.

```
typedef Anchor=
                objAnchor [H3d HMat3d S I]
              | posAnchor [S [I I I] [I I I]];;
```

Cette liste est composée de deux types d'éléments :

- `objAnchor` : un tuple (objet3d, matériau, nom, flag de visibilité)
- `posAnchor` : un tuple (nom de la position, vecteur, angles)

L'ancre est donc littéralement le point d'accroche d'une fonctionnalité à la scène 3d. Par exemple, un module de rotation a besoin d'un ensemble d'objets à faire tourner, un module de déplacement a besoin d'objets et de trajectoires, un module de clignotement a besoin d'une liste de matériaux, ... Certaines fonctionnalités n'ont pas besoin d'ancre : par exemple, un module affichant un logo en surimpression dans un coin de l'image 3d.

Le paramètre "`anchor`" d'une instance contient le nom de l'ancre associée à l'instance.

Cela peut être le nom d'une ancre définie dans l'éditeur du module, ou bien une définition directe de la forme : `strbuild ("#"::nom_H3d::nom_HMat3d::nil)::nil`

Dans ce dernier cas, l'ancre est une liste à un seul élément.

14.10.3. Plugins

14.10.3.1. Généralités

La structure Ob représente un objet, c'est-à-dire l'instance d'une certaine classe. Le rôle des plugins est de décrire les classes. Il y aura exactement une classe définie pour chaque plugin. Un plugin ne définissant pas de classe est sans intérêt.

Le plugin est décrit par un fichier *.plug ressemblant beaucoup au format *.dmc. On y trouve les mêmes champs :

- `name` : nom du plugin
- `help` : fichier texte d'aide
- `serverNeeded` : fichiers nécessaires au plugin serveur
- `serverLoad` : liste des fichiers à compiler successivement pour lancer le serveur
- `clientNeeded` : fichiers nécessairs au plugin client
- `clientLoad` : liste des fichiers à compiler successivement pour lancer le client
- `editorNeeded` : fichiers nécessaires au plugin éditeur
- `editorLoad` : liste des fichiers à compiler successivement pour lancer l'éditeur
- `version` numéro_de_version numéro_de_sous-version

A la différence du fichier dmc, les lignes `server*` sont facultatives, de même que les lignes `client*`. Mais un plugin qui n'aurait ni ligne `server*`, ni ligne `client*` n'aurait aucun intérêt.

14.10.3.2. Plugin interne

L'éditeur C3d détermine les plugins nécessaires aux instances définies dans l'éditeur. Au lancement du module C3d, les plugins nécessaires sont chargés. Attention, pour être reconnu par l'éditeur C3d, le répertoire contenant le fichier *.plug doit se trouver dans le répertoire *'Dms/3d/Plugins'*

Une fois le plugin chargé, la fonction `IniPlug` est lancée (l'équivalent du `IniDmi` pour les modules dmi), avec en argument le nom du fichier *.plug* : ce fichier est immédiatement utilisable, puisque son téléchargement est un préalable au lancement du plugin.

Le plugin appelle typiquement la fonction suivante :
```
I PlugRegister(class S new fun [Ob] I close fun[] I
```

Cette fonction enregistre la classe, avec la fonction appelée lors de la création d'un objet, et la fonction appelée avant la destruction du module C3d.

Cette fonction est la même sur le serveur et sur le client.

Attention, si des instances de la classes sont déjà dans le module, la fonction `new` sera appelée avant que la fonction `PlugRegister` ne termine.

Le plugin a accès à une Api particulière.

Il a accès à une variable globale `'thisplug'` qui est un pointeur vers lui-même de type Plug (de même qu'un module à accès à la variable `'this'` qui est un pointeur vers lui-même de type DMI)

```
[[S r1] r1] PLUGparam plugin Plug
```

Retourne les paramètres du plugin (sortes de variables de classe)

`[Plug r1] PLUGlist`
Retourne la liste des plugins

`S PLUGfile plugin Plug`
Retourne le nom du fichier .plug

`S PLUGclass plugin Plug`
Retourne la classe du plugin

Flags d'information plugin : un plugin peut, côté client, donner quelques informations sur lui-même. Cela se fera généralement lors du `IniPlug`, une fois pour toutes. Ces flags sont des composés des masques suivants :

- `PLUGIN_ONLINE_EDITING` le plugin est prévu pour être édité en ligne
- `PLUGIN_WHOLE_OBJECT` le plugin s'appuie sur une ancre contenant un objet 3d, sans précision du matériau

Les fonctions d'utilisation du flag d'information sont :

`I PLUGinfo plugin Plug`
retourne la valeur courante

`I PLUGsetinfo plugin Plug`
définit une nouvelle valeur

`I PLUGdefineEditor plugin Plug callback fun [ObjWin H3d HMat3d S`
définit la callback de création d'éditeur

`fun [] S fun PLUGstartEditor plugin Plug window ObjWin 3dhandler H3d material HMat3d parameters S`
lance l'éditeur associé à un plugin avec des paramètres initiaux

Dans l'éditeur, on appelle également le fichier `IniPlug` avec en paramètre le nom du fichier **.plug*. Le plugin appelle alors typiquement la fonction suivante :

`I PlugRegister class S save fun [ [Inst r1] ] [[S r1] [S r1] [[S r1]r1] [[S r1]r1]] close fun[] I openedit fun[ObjWin S] I closeedit fun[] S`
Cette fonction `save` est appelée juste avant la sauvegarde du fichier. En entrée, elle récupère la liste des instances définies dans l'éditeur, dont la classe correspond. Cette fonction n'est pas appelée si cette liste est vide.

La structure `Inst` est définie de la manière suivante :

```
struct Inst=[nameInst:S,classInst:S,anchorInst:S,paramInst:S]mkInst;;
```

La fonction `save` doit retourner un tuple de deux listes de mots, et de deux listes de listes de mots, soit un tuple de quatre éléments :

- le premier élément est une liste de fichiers à ajouter à la ligne `'registerF'` du bloc dmi
- le second élément est une liste de fichiers à ajouter à la ligne `'register'` du bloc dmi
- le troisième élément est une liste de lignes à ajouter à la fin du bloc dat (format `strextr`)
- le quatrième élément est une liste de lignes à ajouter à la fin du bloc dmi (format `strextr`)

Attention, les fichier **.plug* et *clientNeeded* sont automatiquement enregistrés comme fichiers téléchargeables : il est inutile de les retourner dans la fonction save.

Dans le cas général, il suffira d'ajouter la ligne `'plugin fichier_*.plug'` au bloc dat.

On pourra ajouter au fichier dmi de nouveaux événements, de nouvelles actions ainsi que de nouvelles zones.

Si la fonction `save` n'est pas définie (`nil` dans la fonction `PlugRegister`), un fonction `save` standard sera appelée qui retourne le tuple suivant :

```
      [
  nil /* registerF */
  nil /* register */
  ("plugin"::fichier_plugin::nil)::nil /* supplemental Dat */
  nil /* supplemental Dmi */
 ]
```

Cela suffit la plupart du temps

14.10.4. Exemples

14.10.4.1. Exemple 1 : module rotate

Dans cette exemple, on donne la version "longue" de l'éditeur, version équivalente à une absence de définition de la fonction `'save'`.

Fichier *Dms/3d/Plugins/Rot/rot.plug*

```
name Rotate
help Dms/3d/Plugins/Rot/rot.help
clientNeeded ./rotc.pkg
clientLoad ./rotc.pkg
editorNeeded ./rotedit.pkg
editorLoad ./rotedit.pkg
version 2 0
```

Fichier *Dms/3d/Plugins/Rot/rotc.pkg*

```
/* Rotate Plugin - DMS - March 00 - by Sylvain HUET */

typeof class=S;;

fun rotobj2(x,v)=
 match x with
  (objAnchor [h _ _ _] -> M3rotateObj session h v)
 |(_->nil);;

fun rotobj(o,v)=apply_on_list ObAnchor o @rotobj2 v;;

fun newOb(o)=
 let hd UgetParam ObUi o "angular" -> s in
 let nth_char s 0 -> a in
 let if (a>=48 && a<58)||a=='- then ['y 109*atoi s]
  else [a 109*atoi substr s 1 1000] ->[v i] in
 let if v=='x then [0 i 0]
  else if v=='z then [0 0 i]
  else [i 0 0] -> v in
   ObCbAnim o mkfun2 @rotobj v;
```

```
 0;;

fun IniPlug(file)=
 set class=getInfo strextr _getpack _checkpack file "name";
 PlugRegister class @newOb nil;
 0;;
```

Fichier *Dms/3d/Plugins/Rot/rotedit.pkg*

```
/* rotedit.pkg : editeur du plugin rot */

typeof plugin=S;;

proto save=fun [ [Inst r1] ] [[S r1] [S r1] [[S r1]r1] [[S r1]r1]];;

fun save(l)=
[
  nil /* registerF */
  nil /* register */
  ("plugin"::plugin::nil)::nil /* supplemental Dat */
  nil /* supplemental Dmi */
 ];;

fun IniPlug(file)=
 set plugin=file;
 PlugRegister (getInfo strextr _getpack _checkpack file "name")
  class nil nil nil nil;;
```

14.10.4.2. Exemple 2 : module test

Le module '*test*' illustre le plugin serveur et les possibilités de communication entre objet serveur et objet client. Le principe est le suivant : en cliquant sur un objet (premier élément de l'ancre de l'instance), l'utilisateur en change aléatoirement la couleur flat. Ce changement est global : il s'applique chez tout le monde. Ce changement est persistant : le serveur conserve en permanence la couleur courante, et la transmet aux nouveaux clients.

Le système de message est le suivant :

- le serveur transmet la couleur par un message `'setFlat'`
- le client demande la couleur courant par un message `'color?'`
- le client indique qu'il clique sur l'objet par un message `'click'`

Les fonctions de communication utilisées sont celles des UserInstance, utilisées comme surclasses de la structure Ob.

Fichier *Dms/3d/Plugins/Test/test.plug*

```
name Test
help Dms/3d/Plugins/Test/test.help
serverNeeded ./tests.pkg
serverLoad ./tests.pkg
clientNeeded ./testc.pkg
clientLoad ./testc.pkg
editorNeeded ./testedit.pkg
editorLoad ./testedit.pkg
version 2 0
```

Fichier *Dms/3d/Plugins/Test/tests.pkg*

```
/* Rotate Plugin - DMS - March 99 - by Sylvain HUET */

typeof class=S;;

fun cbcomm(ui,cli,action,param,z)=
 let z->[o col] in
 if !strcmp action "click" then
 (set col=(rand&255)+((rand&255)<<8)+((rand&255)<<16);
  mutate z<-[_ col];
  UsendCli this nil ui "setFlat" itoa col)
 else if !strcmp action "color?" then
  UsendCli this cli ui "setFlat" itoa col
 else nil;;

fun newOb(o)=
 UcbComm this ObUi o mkfun5 @cbcomm [o 1];
 0;;

fun IniPlug(file)=
 set class=getInfo strextr _getpack _checkpack file "name";
 PlugRegister class @newOb nil;
 0;;
```

Fichier *Dms/3d/Plugins/Test/testc.pkg*

```
/* Rotate Plugin - DMS - March 99 - by Sylvain HUET */

typeof class=S;;

fun appFlat(x,col)=
 match x with
  (objAnchor [_ m _ _] -> M3setMaterialFlat session m col)
 |(_->nil);;

fun applyFlat(o,col)=
 apply_on_list ObAnchor o @appFlat col;
 0;;

fun cbcomm(ui,action,param,o)=
 if !strcmp action "setFlat" then
  applyFlat o atoi param
 else nil;;

fun cbclick(o,h,m,i)= UsendSrv this ObUi o "click" nil;;

fun newOb(o)=
 UcbComm this ObUi o mkfun4 @cbcomm o;
 match hd ObAnchor o with
  (objAnchor [h _ _ _] -> ObSetMain o h)
 |(_->nil);
 ObCbClick o @cbclick;
 UsendSrv this ObUi o "color?" nil;
 0;;

fun IniPlug(file)=
 set class=getInfo strextr _getpack _checkpack file "name";
 PlugRegister class @newOb nil;
 0;;
```

Fichier *Dms/3d/Plugins/Test/testedit.pkg*

```
/* edit.pkg : editeur du plugin */

fun IniPlug(file)=
 PlugRegister (getInfo strextr _getpack _checkpack file "name")
  class nil nil nil nil;;
```