SCOL-MAGMA : un tutorial

Sylvain HUET

2 Janv 1997

1 Architecture de base

1.1 Présentation

Scol est un schéma de communication sur internet. Il permet de réaliser des applications réparties en petits modules communiquant via le protocole Tcp-Ip. La fonction principale d'un module est de communiquer avec au moins un autre. Les fonctions d'affichage, quoique sophistiquées, ne sont que des effets de bord du système. Scol n'a pas vocation à devenir un standard graphique, mais un standard de communication directe.

Les deux principales applications actuelles d'internet, que sont le courrier électronique et le Web, sont des moyens de communications basés sur le mode de la diffusion asynchrone : l'information est stockée sur un disque, puis est diffusée vers un ou plusieurs utilisateurs. La communication est à l'initiative unique du client, le serveur n'ayant pas de réel rôle actif, puisqu'il se contente de transmettre les données expressément nommées par le client.

Avec Scol, il s'agit d'établir un lien direct et permanent entre deux noeuds du réseau. Par ce lien, ce ne sont pas seulement des fichiers de données qui circulent, mais également des commandes d'exécution de routines : chacun des deux noeuds autorise l'autre à commander une partie du programme qu'il abrite.

1.2 Universal Scol Machine

L'objet de base de Scol est appelé Universal Scol Machine (USM). C'est concrêtement le programme unique qui se trouve sur chaque noeud Scol du réseau. Ce programme se nourrit de fichiers de code Magma (langage spécifique décrit plus loin), et peut se comporter comme serveur, comme client, ou les deux à la fois.

1.2.1 le canal et son environnement

La machine Scol est organisée autour des liaisons réseau, appelés "canaux", qu'elle abrite. Ces canaux sont des liaisons de type socket Tcp-Ip. Chaque canal est lié à un environnement donné.

L'environnement d'un canal est en fait une liste de fonctions et de variables. Par un mécanisme qui sera décrit plus loin, certains éléments d'un environnement peuvent être partagés entre plusieurs canaux.

1.2.2 fonctionnement de base

La machine passe son temps à attendre qu'un message lui parvienne par un de ses canaux.

Lorsqu'un message est arrivé, celui est découpé sous la forme "Commande Arguments", les arguments étant soit des entiers, soit des blocs d'octets. La routine associée à la Commande est recherchée dans l'environnement du canal, puis exécutée.

1.2.3 aspect seveur

La machine Scol peut tendre des liens réseau (de type socket Tcp-Ip) vers d'autres machines Scol. Pour que ceci soit possible, il faut qu'une machine Scol puisse ouvrir au moins une socket serveur.

Lorsqu'une autre machine Scol ouvre une liaison vers ce serveur, un canal est créé, avec un environnement particulier, défini lors de la création du serveur.

1.2.4 aspect interface

Les interfaces fonctionnent sur un mode sensiblement différent. Pour ce qui est des sorties, l'interface se comporte comme un effet de bord : une routine effectue un calcul et retourne un résultat, en affectant au passage l'interface graphique ou autre.

Les entrées de type événement (clic souris, touche enfoncée) sont un problème différent. La détection d'un événement entraîne l'exécution d'une routine dans un certain environnement.

L'implémentation première de Scol se propose donc de définir des USM spéciales qui concentrent l'aspect interface, en étant capable d'écouter sur des fils de natures différentes : canaux et événements graphiques ou autres.

L'avantage de ce système est de simplifier la tâche de l'utilisateur final, en ne lui donnant accès qu'aux USM classiques ; ce cette manière en effet, l'utilisation de l'interface se résume au même mécanisme de communication que celui utilisé pour dialoguer avec les autres machines Scol. L'autre avantage de cette solution est de reconstituer l'architecture Xwindow, avec la notion de serveur graphique et de délocalisation du client.

1.2.5 mort de la machine Scol

Une machine Scol ne réagit que dans deux situations : réception d'un message sur un canal et connexion d'un client sur le serveur. En conséquence, si une machine Scol n'a ni canal ouvert ni serveur défini, elle peut être considérée

comme morte : rien ne peut plus se produire pour elle. La machine Scol détecte un tel état, et s'éteint automatiquement.

2 Magma

Pour que Scol fonctionne, il fallait un langage de programmation permettant plusieurs choses : . une gestion dynamique d'ensembles de routines . une gestion d'environnements . un code mobile et portable

Ce langage s'appelle Magma. C'est un langage fonctionnel compilé en bytecode et exécuté sur la machine virtuelle contenue dans les USM.

Le bytecode est un langage à pile, fonctionnant sur une bande mémoire gérée par un système de type Garbage Collector.

2.1 syntaxe de Magma

$Magma$	=	M			|	$MMagma$
M	=	**typeof**V=$Type$;;			|	**typeof**F=$Type$;;
	|	**var**V=A';;				
	|	**fun**$F(X)$=P;;				
	|	**typedef**N_T=C;;				
	|	**struct**N_T=[$Champs$]$Constr$;;				
	|	**defcom**Com=$string\{I,S,P\}*$;;				
	|	**defcomvar**$Com\,V$=$\{I,S,P\}*$;;				
$Type$	=	B	|	$\mathbf{u}_n$	|	$\mathbf{r}_n$
	|	**tab**$Type$	|	[$Type*$]	|	**fun**[$Type*$]$Type$
B	=	Type de base				
$\mathbf{u}_n$	=	variable liée				
$\mathbf{r}_n$	=	récursion de niveau n				

P	=	A	|	A**;**P		
A	=	A_1	|	A_1**&&**A	| $A_1 \| \| A$	
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	|	$A_4 + A_4$	| $A_4 - A_4$	
	|	$A_4 + .A_4$	|	$A_4 - .A_4$		
A_4	=	A_5	|	$A_5 * A_5$	| A_5 / A_5	
	|	$A_5 * .A_5$	|	$A_5 / .A_5$		
A_6	=	T	|	$-A_6$	| $\tilde{}A_6$	
T	=	*int*	|	**'***char*	| **nil**	| *string*
	|	(P)	|	$V(.T)*$	| FL_F	| $[L]$
	|	**if**A**then**A**else**A	|	**while**A**do**A	| **set**$V(.T)*=A$	| **mutate**$A < -[L']$
	|	**let**$A- > N$**in**A	|	**@**F	| **exec**A**with**A	| *ConstrA*
	|	*Constr0*	|	**match**A**with***Case*	| $V(.N_C)*$	| **set**$V(.N_C)*=A$
L_F	=	$A...A$		$narg(F)$fois		
L	=	$A*$				
L'	=	$\{\_, A\}*$				
X	=	*rien*	|	X'		
X'	=	N'	|	N', X'		
N	=	N'	|	$[\{\_, N\} *]$		
A'	=	*int*	|	**'***char*	| **nil**	| *string*
	|	$-int$	|	$[L'']$		
L''	=	$A'*$				
C	=	C'	|	$C'\|C$		
C'	=	*ConstrType*	|	*Constr0*		
Champs	=	*Champ*	|	*Champ*, *Champs*		
Champ	=	N_C : *Type*				
Case	=	*Case'*	|	*Case'*|*Case*	| $(\_- > A)$	
Case'	=	$(ConstrN' - > A)$	|	$(Constr0 - > A)$		
V	=	variable				
F	=	fonction				
N'	=	variable liée				
Constr	=	constructeur				
Constr0	=	constructeur0				
N_C	=	nom de champ				
Com	=	constructeur de communication				
ComV	=	constructeur de communication variable				
int	=	entier				
char	=	caractère				
string	=	chaîne				

Les entiers sont soit :

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

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 :

$\backslash n$: retour chariot
$\backslash$" : guillemet
$\backslash\backslash$: backslash
$\backslash 0$: caractère 0

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.

2.2 syntaxe de type

La syntaxe de types a été définie dans le paragraphe précédent. Voici quelques précisions :

Exemples :

- le type d'une liste d'entiers est : "[I r1]"
- le type d'une fonction d'addition est : "fun [I I] I"

Les types standard sont :

I : int
S : string
F : float
Chn : canal SCOL
Arg : chaîne d'arguments
Comm : communication

Attention : la constante nil n'a pas de type défini.

Un type est valide s'il ne contient pas de variable libre : "u0" et "fun [ ] u0" ne sont pas valides.

2.3 Explications de base

Il y a deux types principaux de déclarations : les déclarations de fonctions et les déclarations de variables (fun et var). Il est possible et parfois nécessaire de déclarer au préalable le type d'une fonction ou d'une variable (typeof). Ceci est

utile lorsque l'on déclare une variable initialisée à nil, ou lorsqu'on déclare un prototype pour réaliser des fonctions croisées (f appelle g et g appelle f).

Deux structures sont disponibles : le tuple (entre crochets) qui est une table d'éléments de types indépendants, et le tableau qui est une table d'éléments ayant tous le même type.

2.3.1 Manipulation des tuples

Le terme [L] construit un tuple à partir des différentes expressions de L. 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 grace à la fonction let : let [1 2 nil [3 4]] $->$ [a _ b c] in A dans A, les variables a,b et c valent respectivement 1, nil et [3 4].

On modifie un ou plusieurs champs d'un tuple grace à la fonction mutate : let [1 2 nil [3 4]] $->$ tupletest in mutate tupletest $<-$ [_ 5 6 _] La fonction mutate remplace ici les valeurs 2 et nil du tuple par respectivement 5 et 6, sans toucher aux autres champs. 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.

2.3.2 Manipulation des tableaux

La création d'un tableau se fait grace à 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).

2.3.3 Valeur de certaines fonctions

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

La fonction "if Condition then Vrai else Faux" calcule la condition (qui doit être un entier). Si cet entier est vrai (c'est-à-dire différent de 0), l'expression Vrai est calculée, et le résultat de la fonction "if" est le résultat de l'expression Vrai . Dans le cas contraire c'est l'expression Faux qui est calculée.

Exemple : let if 1=2 then 3 else 4 $->$ x in ... dans cette expression, x prend la valeur 4.

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).

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.

2.3.4 Conditions

Les conditions sont des expressions 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)

2.3.5 Exemples de programme

calcul de la taille d'une liste quelconque : chaque maillon de la liste est un tuple de taille 2 dont le deuxième élément est le maillon suivant.

```
/* routine calculant la longueur d'une liste qquelconque */

fun sizelist(l)=
  if l=nil then 0
  else let l -> [_ n]
       in 1 + sizelist n;;
```

le type de sizeliste est “fun [[u0 r1]] I”.

Quicksort :

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

/* tri quicksort */
fun dividelist (x,p,r1,r2)=
  if p=nil then [r1 r2]
  else
    let p->[a n]
    in if x<a
       then dividelist x n [a r1] r2
```

```
       else dividelist x n r1 [a r2];;

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

2.4 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 traîte un objet de type U grace 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.

Il est recommandé de mettre toujours une dernière ligne *par défaut* car en cas d'oubli d'un constructeur, l'interpréteur magma détectera une erreur de même importance qu'une division par 0 ou que l'utilisation d'un index trop grand dans un tableau.

Exemple plus évolué :

```
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));;
```

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.

2.5 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 l'exemple précédent, 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, on ne peut avoir deux structures qui utilisent un même nom de champ.

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.x;;
```

Cette fonction retourne 1.

2.6 Manipulation des fonctions

Magma 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, an 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.

2.7 librairie standard

size : 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.

strlen : fun [S] I

retourne la taille d'une chaîne de caractères.

strcat : fun [S S] S

concatène deux chaînes de caractères (les deux chaînes initiales ne sont pas modifiées)

strcatn : fun [[S r1]] S

concatène une liste de chaînes de caractères (équivalente, mais en plus efficace, à la fonction précédente répétée n fois)

strcmp : fun [S S] I

compare deux chaînes de caractères (en utilisant la fonction C standard)

listtostr : fun [[I r1]] S

transforme une liste d'entiers en chaîne de caractères : chaque maillon donne un caractère. Le premier maillon donne le premier caractère.

strtolist : fun [S] [I r1]

fonction inverse de la précédente.

atoi : fun [S] I

interprète une chaîne de caractères comme un entier.

itoa : 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, et le second en donne la taille.

strdup : fun [S] S

crée une copie en mémoire d'une chaîne de caractère.

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)

rand : fun [] I

retourne un entier aléatoire, compris entre 0 et 65535

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

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.

3 Environnements

3.1 Définition

Un environnement est une liste de variables et de fonctions. Pour exécuter une certaine commande, par exemple “say ”hello world””, il faut trouver la fonction say dans l'environnement, et l'exécuter avec comme argument “hello world”. Dans une machine SCOL, chaque canal dispose d'un environnement qui lui est propre, dans lequel sont exécutées les commandes qui parviennent par ce canal.

Lorsqu'on compile un fichier écrit en Magma, on spécifie un environnement, qui est l'environnement de compilation : le fichier peut faire référence aux fonctions et variables de l'environnement, et les fonctions et variables définies dans le fichier s'ajoutent à l'environnement de compilation.

L'environnement minimal est un ensemble de fonctions permettant de charger et de compiler un fichier Magma.

3.2 Partage d'environnements en SCOL

Plusieurs canaux peuvent avoir à partager un certain nombre de fonctions et de variables, ce qui permet de faire le lien entre les canaux. Cela signifie qu'une partie de leur environnement est commune : les deux listes de fonctions et de variables se rejoignent à partir d'un certain maillon.

Pour rendre ce partage aisé, une notion d'application est introduite en SCOL : l'application est le sous-environnement de l'environnement d'un canal qui peut être partagé avec d'autres canaux. On parlera d'environnement canal et d'environnement application.

Lorsqu'un nouveau canal est créé, il l'est dans l'environnement d'un certain canal C. Le nouveau canal D hérite de l'environnement application de C. L'environnement canal de D est initialisé à l'environnement application : à la création de D, l'environnement de D est exactement l'environnement application de C. Les fichiers compilés par la suite dans l'environnement de D se placeront dans l'environnement canal de D et seront donc inaccessibles depuis le canal C : les fonctions Magma ont une portée statique, c'est-à-dire définie à la compilation.

3.3 Fonctions Magma de gestion des environnements et des canaux

Le type Chn représente un canal.

_script : fun [S] I

exécution d'un script dans l'environnement courant. Chaque ligne du script est de la forme commande-arguments : la portée est définie dynamiquement. Le

résultat est 0 si tout s'est bien passé (pas d'erreur d'exécution de type division par zéro ou autre). Un commande inconnue, ou un mauvais typage des paramètres ne sont pas considérés comme des erreurs : la commande est simplement sautée.

_load : fun [S] I

chargement d'un fichier Magma (appelé aussi package), dont le nom est passé en paramètre (voir plus loin la gestion des fichiers).

_loadhard : fun [S] I

chargement d'un package *en dur*, c'est-à-dire un package de fonctions écrites en C, dans la machine SCOL (principalement des fonctions d'entrée-sortie).

_openchannel : fun [S S] Chn

création d'un canal vers une certaine adresse, et en exécutant un certain script dans l'environnement du nouveau canal. Ce nouveau canal hérite de l'environnement application du canal "père".

_resetchannel : fun [] I

réinitialisation de l'environnement du canal : celui-ci redevient égal à l'environnement application.

_newappli : fun [] I

réinitialise l'environnement complet d'un canal : l'environnement application devient NIL, et l'environnement canal se résume aux commandes de base permettant notamment le chargement d'un package. Avant de pouvoir exécuter la commande _openchannel dans l'environnement de ce canal, il faudra d'abord avoir redéfini un environnement application grace à la commande suivante.

_setappli : fun [] I

définit l'environnement application d'un canal. Ceci n'est possible qu'une fois, après un _newappli. C'est indispensable pour pouvoir ouvrir un autre canal avec _openchannel (puisque le nouveau canal hérite alors de l'environnement application).

_channel : fun [] Chn

récupère le canal courant.

_closechannel : fun [] I

ferme le canal courant.

_killchannel : fun [Chn] I

ferme un canal appartenant à la même application.

4 Communications

Une fois le canal créé, des messages peuvent transiter dans les deux sens. Les messages qui circulent sont toujours de la forme "commande arguments". 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 receptrice 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 ignoré : seuls compteront les effets de bord.

Pour utiliser le canal dans le sens de l'émission, le langage magma offre deux solutions. On prendra l'exemple suivant : on veut envoyer sur le canal ch la commande *foo* avec trois paramètres : deux entiers 123 et 345 et une chaîne "bar".

4.1 Envoi d'un message par le _say

En plus des types de base I, S, F, Chn, le type Arg est défini en standard. Sa définition exacte est :

```
typedef Arg =
   ArgI [I Arg]
 | ArgS [I Arg];;
```

La fonction _say est de type "fun[Chn S Arg] I". Elle retourne 0 en cas de succès. L'exemple précédent se code donc :

```
...
  _say ch "foo" ArgI[123 ArgI[345 ArgS["bar" nil]]];
...
```

Mais rien n'empêche le programmeur d'écrire par inadvertance la ligne suivante, sans que le compilateur puisse détecter le moindre problème :

```
...
  _say ch "foo" ArgI[123 ArgS["bar" nil]];
...
```

Pour aller plus loin, et pour gagner en clareté d'écriture, il est préférable d'utiliser le _on.

4.2 Envoi d'un message par le _on

Avec le _on, l'exemple précédent devient :

```
defcom X = foo I I S;;

...
  _on ch 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 ici 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".

Le _on est une fonction de type "fun[Chn Comm] I".

Pour simplifier l'écriture, on peut encore écrire :

```
defcom foo = foo I I S;;

...
  _on ch foo [123  234 "bar"];
...
```

Il existe une variante avec *defcomvar*. Cette variante est notamment intéressante pour l'utilisation des call-back. 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 solution est ici :

```
defcomvar Y = I;;

fun call(s,i)=
  _on ch 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 :

```
defcom Y = I I S;;

...
  _on ch Y "foo" [123  234 "bar"];
...
```

5 Gestion des fichiers

SCOL autorise en standard un accès restreint aux fichiers de la machine. Pour cela, l'utilisateur définit un ou plusieurs répertoires particuliers appelés *partitions SCOL*.

5.1 Types de fichier en SCOL

SCOL offre deux types de sauvegarde : la sauvegarde normale et la sauvegarde signée.

Sauvegarde normale

Dans ce mode, l'utilisateur donne le nom du fichier à sauvegarder, sans restriction. En revanche, c'est le système qui détermine le répertoire de sauvegarde, choisi parmi les partitions SCOL disponibles. 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 et le underscore.

Sauvegarde signée

Dans ce mode, l'utilisateur donne un *nom en clair*, composé de caractères alphanumériques, de points et de underscore, 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 Partitions SCOL

Une partition SCOL est un répertoire contenant lui-même autant de sous-repertoires que de types de fichiers (un pour le type normal, et un par type

de signature). Une partition est définie comme *Read-Only* ou *Writable*. Dans ce dernier cas, on peut préciser un quota, éventuellement infini.

Toute machine SCOL utilise un fichier **usmpack.ini** qui donne la liste des partitions de la machine. C'est un fichier texte dont chaque ligne définit une partition : un chemin, suivi éventuellement d'un nombre en décimal indiquant le quota.

L'ordre de définition des partitions est important : la recherche d'un fichier s'effectue en effet dans le même ordre.

```
# exemple de fichier usmpack.ini

# les lignes commencant par '#' sont des lignes de commentaire

# premiere partition : 4Mo
/scol/pack/        4096

# trois partitions read-only :
/scol/archivage/
/cdrom1/
/cdrom2/
```

En cas d'absence de fichier usmpack.ini, les fichiers seront écrits et recherchés dans le répertoire courant de la machine.

5.3 Librairie Magma

Pour utiliser les fichiers, le type 'P' a été créé (P pour *Path*). Il reste globalement opaque à l'utilisateur.

_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*.

_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.

_load : fun [S] I

cette fonction a déjà été décrite, elle charge et compile un fichier contenant du code Magma. En fait le fichier est recherché dans les partitions SCOL de la machine.

_getlongname : fun [S_1 S_2 S_3] S

calcule le nom complet d'un fichier :

- S_1 est le fichier à sauvegarder
- S_2 est le nom en clair
- S_3 code le type de signature, en reprenant le caractère spécifique
  - "" : pas de signature
  - "#" : premier type de signature

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

```
/* Exemple de manipulation de fichiers */
/* lit un fichier foo.pkg et le recopie sous le nom 'bar#signature',
   en v\'erifiant le r\'esultat de chaque \'etape */

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

5.4 Fonctions avancées de gestion de fichier

La fonction _getpack décrite précédemment permet de charger dans une chaîne de caractères l'ensemble du fichier. Si ce fichier est un fichier de configuration

comportant, en Ascii, des lignes de mots, il est intéressant de disposer d'outils d'analyse puissants, et de ne pas laisser à l'utilisateur le soin de refaire l'analyse syntaxique.

Le langage Magma intègre deux fonctions complémentaires d'analyse, permettant de passer d'un texte quelconque à une double-liste de mots, et inversement.

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+espace pour n'importe quel code ascii

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

strbuild [[[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 \.