An anesthetizing algorithmic language
16 juin 2004

Table des matières

Chapitre 1

Introduction

1.1 SpécialK, an anesthetizing algorithmic language

[Special K] has both analgesic and amnesic properties and is associated with less confusion, irrationality, and violent behavior
(Source : The National Institute on Drug Abuse (NIDA))

SpécialK est une normalisation du langage fonctionnel à base de règles créé et utilisé par Emmanuel Kounalis (d'où le *K*) en cours d'algorithmiques en 2e année de DEUG MIAS et en license d'informatique à l'université de Nice. Ce langage a pour but l'enseignement de l'informatique et est conçu pour ecrire des algorithmes simples de manière concise avec manipulation simple de listes, arbres, ou tableaux. Le développement d'applications et l'orientation industrielle n'est pas un but recherché.

1.2 SpécialK is a SpécialK to Scheme compiler

DrScheme est à l'origine un environnement de programmation pour le langage Scheme. Cependant, il est maintenant possible d'utiliser l'ensemble des fonctionnalités de ce logiciel pour plusieurs langages. Entre autres, il propose un éditeur de texte éprouvé, coloration syntaxique, gestion de l'exécution et un profileur.

Le *plugin* SpécialK vous permet de tester des algorithmes en SpécialK sans avoir à les traduire en Scheme - il le fait pour vous ! Vous utilisez DrScheme comme à l'accoutumée, mais à la place de taper du Scheme, vous tapez du SpécialK.

SpécialK est un logiciel libre diffusé sous les termes de la GNU General Public License.

1.3 SpécialK is an experimental langage for DrScheme

Depuis quelque temps, DrScheme est une plate-forme multi-langages, à la manière d'Emacs pour l'édition, mais également pour l'exécution. La version 207 est fournie entre

autres avec les langages Algol60 et ProfessorJ (Java). SpécialK se rajoute à cette liste.

```
;;;;;;;;;;;;;;
;   Scheme   ;
;;;;;;;;;;;;;;

; Sum of an array's elements

(define T (vector 1 2 3 4 5 6))

(define (sum T i)
  (if (>= i (vector-length T))
      0
      (+ (vector-ref T i) (sum T (+ i 1)))))

(sum T 0)
```

```
/******************/
/*      Java      */
/******************/

/* Sum of an array's elements */

class Test{

public static void main(String[] args){
int[] T = {1,2,3,4,5,6};
int r = 0;
for(int i = 0; i < T.length; i++)
r += T[i];
}
}
```

```
/******************/
/*    SpécialK    */
/******************/

/* Sum of an array's elements */

T <- {1,2,3,4,5,6}.

i >= tab-len(T) -> sum(T,i) = 0;
i <  tab-len(T) -> sum(T,i) =  T[i] + sum(T,i+1).
```

```
sum(T,0).
```

1.4 À propos de ce document...

Ce document a deux buts.

Tout d'abord, il s'adresse à tout utilisateur connaissant les bases de la programmation et voulant programmer en SpécialK. Il contient une description du langage ainsi qu'une série d'exemples. Ce document n'aborde pas les techniques de programmation en SpécialK autrement qu'au travers des exemples donnés. SpécialK permettant d'utiliser toutes fonctions de bibliothèque de MzScheme, l'utilisateur devra se référer au manuel de MzScheme pour se documenter sur leur fonctionnement. On considère que l'utilisateur connaît les bases du langage Scheme.

Ensuite, le chapitre 5 peut servir de base pour le spécification informelle du langage. Il décrit sa structure et le comportements qu'il doit avoir.

1.5 Contacter les auteurs

Si vous découvrez un bug, ou voulez simplement prendre contact avec les auteurs de ce logiciel, voici leurs adresses respectives :

Sylvain Beucler	`beuc@beuc.net`
Julien Charles	`charlesju@wanadoo.fr`
Pierre Châtel	`admin@chatelp.org`
Cyril Rodas	`cyril.rodas@wanadoo.fr`

Chapitre 2

Prise en main

2.1 Installation

SpécialK nécessite la version 207 de DrScheme. Il est distribué le plus couramment sous la forme d'un fichier `.plt`.

Pour l'installer, cliquez sur *Fichier / Installer un fichier .plt...* puis sélectionnez le fichier `specialk.plt`.

Nous avons décidé de faire en sorte que l'installation soit possible pour un utilisateur ordinaire (sans privilèges administrateur) ; aussi `specialk.plt` s'installe par défaut dans le répertoire de collections de l'utilisateur, et non celui du système où l'utilisateur n'a pas forcément accès.

Pour effectuer une installation globale pour tous les utilisateurs de la machine, entrez à la place la commande

```
# setup-plt --all-users specialk.plt
```

dans une console. Les exécutables de DrScheme étant installés par défaut, sous Unix, dans `/usr/local/plt/bin`, aussi vérifiez que `setup-plt` est bien dans votre *`PATH`*.

2.2 Les modes SpécialK

Pour activer SpécialK, utilisez la boîte de dialogue *Configurer le langage* dans le menu *Langage / Sélectionner le langage*. Dans la liste des langages disponibles, un nœud SpécialK propose deux choix :

- *SpécialK Classic* effectue un maximum de tests à la compilation comme à l'exécution, facilitant l'écriture de code propre, mais allourdissant le code produit. C'est le langage de départ et celui qui est définit dans le chapitre 5.
- *SpécialK Light* en revanche supprime ces tests, produisant ainsi un code Scheme plus rapide. À utiliser une fois l'algorithme finalisé.

Sélectionnez le niveau de langage qui vous le convient puis validez.

SpécialK utilise la langue fournie par l'environnement DrScheme pour afficher ses messages. Actuellement, les langues disponibles sont l'anglais et le français.

2.3 Test

Dans la fenêtre de définitions, entrez :

```
fac(0) = 1;
n > 0 -> fac(n) = n * fac(n - 1).
```

Cliquez sur *Exécuter*, puis évaluez l'expression `fac(10).` au top-level :

```
Bienvenue dans DrScheme, version 207.
Language: SpécialK Classic.
> fac(10).
3628800
```

Chapitre 3

Un survol de SpécialK

Cette partie est un survol rapide de la structure et de l'exécution d'un programme en SpécialK. Tout les points abordés seront détaillés dans le chapitre 5.

3.1 Premier programme

Cette fois-ci, ce ne sera pas *"Hello World"*... Ce premier algorithme, un tri sur liste, va servir à décrire la structure d'un programme.

```
/* Commentaire : tri par insertion */

/* Définition de la fonction insert/2 (une clause par ligne) */
insert(e,nil) = e:nil;                          /* clause sans garde */
e > c -> insert(e,c:L) = c : insert(e, L);      /* clause avec garde */
e <= c -> insert(e,c:L) =  e : c : L.

/* Définition de la fonction tri-insert/2 */
tri-insert(nil) = nil;
tri-insert(c:L) = insert(c,tri-insert(L)).


display("tri par insertion de 23:6:8:43:16:2:9:0:4:nil").   /* 3 */
tri-insert(23:6:8:43:16:2:9:0:4:nil).                       /* 4 */
```

Ce programme aura pour effet d'afficher dans la fenêtre d'interaction de DrScheme la chaîne *tri par insertion de* `23:6:8:43:16:2:9:0:4:nil`, et renverra la liste sous sa forme triée. La compatibilité des structures de SpécialK avec celles de Scheme sera décrite dans la section 5.1.

Un programme en SpécialK est constitué d'expressions et de définitions. Dans l'exemple, deux fonctions sont définies. Une fonction est une suite de clauses séparées

par le symbole ' ;', la dernière étant terminée par '.'. Les clauses ont deux formes possibles : *avec* ou *sans* garde.

Le programme contient ensuite deux expressions, la première étant l'appel à la fonction `display` de MzScheme et la seconde l'appel à la fonction definie plus haut. La forme des expressions est décrite à la section 5.3.

3.2 Exécution du programme

L'exécution s'effectue en cliquant sur le bouton *Exécuter*. Le programme est exécuté de haut en bas. Chaque définition de fonction est ajoutée à l'*environnement global* . Les fonctions n'existent dans cet environnement qu'après leur définition, c'est à dire que l'on ne peut pas appeler une fonction avant sa définition.

À chaque appel de fonction (définie par l'utilisateur), une des clause qui la compose est choisie. Le choix d'une clause est détaillée dans la section 5.4.

Le langage manipule différent types dont le type doublet (ou liste). La section 5.2 traite des types de SpécialK.

3.3 Quelques exemples

3.3.1 Somme des éléments d'une liste

Dans la fenêtre de définitions, entrez :

```
sum(nil) = 0;
sum(c:L) = c + sum(L).
```

Puis au top-level :

```
Bienvenue dans DrScheme, version 207.
Language: SpécialK Classic.
> sum(1:2:3:4:5:nil).
15
```

3.3.2 Somme des éléments d'un tableau

Dans la fenêtre de définitions, entrez :

```
i >= vector-length(T) -> sum(T,i,r) = r;
i < vector-length(T) -> sum(T,i,r) = sum(T,i+1,r+T[i]).
```

Puis au top-level :

```
Bienvenue dans DrScheme, version 207.
Language: SpécialK Classic.
> sum({1,2,3,4,5}, 0, 0).
15
```

3.3.3 Autres exemples

Le dossier `demos/` de la distribution SpécialK contient un jeu de fichiers `.spk` présentant divers algorithmes. Il y a également une implémentation du jeu de la vie.

Chapitre 4

Fonctions

SpécialK vient avec un jeu de fonctions particulièrement utiles lors de la programmation en SpécialK. En voici une description.

Note : Cette partie utilise de nombreux aspects du langage qui sont décrit dans le chapitre 5.

4.1 Fonctions SpécialK par catégories

Constructeur de tableau

make-tab

Accesseurs sur types de données

list-len tab-len

Outils graphiques internes

new-tree-viewer new-tab-viewer display-viewer refresh-viewer

Outils graphiques externes

springgraph vcg xvcg

Conversion d'arbres

tree2dot tree2vcg

Trace de fonctions

trace untrace

Chargement de modules

load-k

Nécessaire à l'exécution (et à oublier si vous n'êtes pas développeur SpécialK)

localized-message match nasty-print

4.2 Fonctions SpécialK par ordre alphabétique

4.2.1 display-viewer

`display-viewer(`*`VIEWER`*`)`

Affiche la représentation graphique *VIEWER*.
Cf. `new-tab-viewer()` et `new-tree-viewer()`.

4.2.2 list-len

`list-len(`*`LIST`*`)`

Renvoie la longueur d'une liste.

Exemple :

```
> list-len(1:2:3:nil).
3
> list-len(1:(2:4:nil):3:nil).
3
```

4.2.3 load-k

`load-k(`*`NOMDEFICHIER`*`)`

Permet d'exécuter un fichier SpécialK séparé.
Les définitions de *NOMDEFICHIER* sont alors accessibles dans le fichier courant. Les définitions de même nom sont écrasées par les nouvelles, sauf celles qui ont un nombre d'arguments différents.

Exemple :

```
load-k("fac.spk").
fac(10).
```

4.2.4 make-tab

`make-tab(`*`DIMENSION`*`...)`

Crée un tableau uni ou multi-dimensionnel dont les éléments sont initialisés à zéro.

Exemple :

```
> make-tab(3).
{0, 0, 0}
> make-tab(3,5).
{{0, 0, 0, 0, 0}, {0, 0, 0, 0, 0}, {0, 0, 0, 0, 0}}
> make-tab(1,2,3).
{{{0, 0, 0}, {0, 0, 0}}}
```

4.2.5 new-tab-viewer

`new-tab-viewer(`*`ARRAY`*`)`

Crée une représentation graphique de tableau (uni ou bidimentionnel) initialisée avec *`ARRAY`*, qui peut être affichée avec la méthode `display-viewer()` ; on peut mettre à jour la valeur du tableau avec `refresh-viewer()`.

Exemple :

```
v <- new-tab-viewer({1,2,{3,4,5}).
display-viewer(a).
refresh-viewer(a, {1}).
```

4.2.6 new-tree-viewer

`new-tree-viewer(`*`TREE`*`)`

Crée une représentation graphique d'arbre initialisée avec *`TREE`*, qui peut être affichée avec la méthode `display-viewer()` ; on peut mettre à jour la valeur de l'arbre avec `refresh-viewer()`.

Exemple :

```
v <- new-tree-viewer(1:2:3:nil).
display-viewer(v).
refresh-viewer(v, 1:2:3:4:nil).
```

4.2.7 refresh-viewer

`refresh-viewer(`*`VIEWER`*`, `*`DATA`*`)`

Change la structure de donnée de l'afficheur *`VIEWER`* en *`DATA`*. Si *`VIEWER`* est affiché, son contenu est mis à jour, permettant de réaliser des animations.

4.2.8 springgraph

`springgraph(`*`TREE`*`[,`*`ARG`*`]...)`

Affiche une image au top-level représentant l'arbre *`TREE`* sous la forme d'un graphe.

Cette procédure traduit tout d'abord l'arbre avec `tree2dot()`. *`TREE`* peut utiliser deux conventions de constructions, de type `r:g:d` ou `r:g:d:nil`. Elle passe ensuite la description de graphe à la commande externe `springgraph` (qui doit se trouver dans le PATH). La commande produit une image PNG qui est la valeur de retour de la fonction, permettant un affichage graphique dans le top-level.

La suite de *`ARG`* permet de fournir des paramètres supplémentaires à `springgraph`. Notamment, `''-s''` permet de définir un facteur de mise à l'échelle.

Le calcul implique un algorithme itératif qui peut s'avérer long sur de grands graphes.

Note : la génération d'image de grande taille ne semble pas s'intégrer à DrScheme qui en général bloque indéfiniment l'exécution.

Plus d'informations sur springgraph :
`http://packages.debian.org/unstable/graphics/springgraph`
`http://www.chaosreigns.com/code/springgraph/`

Exemple :

```
springgraph(1:(2:4:5):(3:6:7):nil, "-s", ".33").
```

4.2.9 tab-len

`tab-len(`*`ARRAY`*`)`

Renvoie la longueur d'un tableau.

Exemple :

```
> tab-len({1,2,3}).
3
> tab-len(make-tab(4,5,6)).
4
```

4.2.10 trace

`trace(`*`FUNCTION`*`)`

Affiche une trace pour chaque appel de la fonction.

`trace` n'est qu'un lien vers la fonction du même nom de la bibliothèque "trace.ss". Veuillez vous référer au chapitre 37 de *PLT MzLib : Libraries Manual* pour plus de détails.

Exemple :

```
load-k("fac.spk").
trace(fac).

> fac(5).
|(fac 5)
| (fac 4)
| |(fac 3)
| | (fac 2)
| | |(fac 1)
| | | (fac 0)
| | | 1
| | |1
| | 2
| |6
| 24
|120
120
```

4.2.11 tree2dot

`tree2dot(`*`TREE`*`)`

Traduit l'arbre *`TREE`* en une représentation de graphe au format .dot. Utilisé par `springgraph()`.

Exemple :

```
> tree2dot("r":"g":"d").
digraph "" {
1 [label="r"]
1 -> 2
2 [label="g"]
1 -> 3
3 [label="d"]
}
```

4.2.12 tree2vcg

`tree2dot(`*`TREE`*`)`

Traduit l'arbre *`TREE`* en une représentation de graphe au format .vcg. Utilisé par `vcg()` et `xvcg()`.

Exemple :

```
> tree2vcg("r":"g":"d").
graph: {
layoutalgorithm: tree
node: { title:"1" label:"r" horizontal_order:1}
edge: { sourcename:"1" targetname:"2" }
node: { title:"2" label:"g" horizontal_order:1}
edge: { sourcename:"1" targetname:"3" }
node: { title:"3" label:"d" horizontal_order:2}
}
```

4.2.13 untrace

`untrace(`*`FUNCTION`*`)`

Supprime la `trace` de la fonction.

4.2.14 vcg

`vcg(`*`TREE`*`)`

Affiche une image au top-level représentant l'arbre *`TREE`*.

Cette procédure traduit tout d'abord l'arbre avec `tree2vcg()`. *`TREE`* peut utiliser deux conventions de constructions, de type `r:g:d` ou `r:g:d:nil`. Elle passe ensuite la description de graphe à la commande externe `xvcg` (qui doit se trouver dans le PATH). La commande produit une image PPM traitée avec `pnmtopng` (également dans le PATH) pour produire une image PNG qui est la valeur de retour de la fonction, permettant un affichage graphique dans le top-level.

Note : la génération d'image de grande taille ne semble pas s'intégrer à DrScheme qui en général bloque indéfiniment l'exécution.

Plus d'informations sur VCG :
`http://packages.debian.org/unstable/graphics/vcg`

Exemple :

```
vcg(1:(2:5:6):(3:7:8):(4:9:10):nil).
```

4.2.15 xvcg

`xvcg(`*`TREE`*`)`

Affiche un arbre à l'aide du programme XVCG, qui permet notamment le zoom.

Cette procédure traduit tout d'abord l'arbre avec `tree2vcg()`. *`TREE`* peut utiliser deux conventions de constructions, de type `r:g:d` ou `r:g:d:nil`. Elle passe ensuite la description de graphe à la commande externe `xvcg` (qui doit se trouver dans le PATH). La fonction rend la main une fois que l'utilisateur quitte XVCG.

Plus d'informations sur VCG :
`http://packages.debian.org/unstable/graphics/vcg`

Exemple :

```
xvcg(1:(2:5:6):(3:7:8):(4:9:10):nil).
```

Chapitre 5

La séance de dissection

5.1 Compatibilité avec MzScheme

SpécialK permet d'appeler la plupart des fonctions de la bibliothèque MzScheme. En effet, tous les types de SpécialK ont un type équivalent en Scheme. Cependant, pour des raisons de cohésion du langage, il n'est pas possible d'appeler les fonctions ayant un ' `!` ' dans leur identifiant, celles-ci étant des fonctions *impératives*, qui briseraient l'aspect fonctionnel pur du langage. D'autre part, ni les formes spéciales et macros, ni les fonctions `eval`, ni les fonctions d'ordre supérieur(c'est à dire les fonctions renvoyant une fonction)ne doivent pas êtres appelées. Cependant, une fonction en SpécialK est comme une *lambda* en Scheme. On peut donc appeler une fonction de MzScheme prenant une fonction en paramètre.

Les fonctions manipulant des types autres que *list*, *vector*, *string* et types numeriques, (c'est à dire tous les types de Scheme qui n'existe pas SpécialK) ne doivent pas êtres appelées.

Par la suite, certaines des fonctions de la bibliothèque MzScheme seront utilisées dans les exemples.

À l'inverse, il est possible d'utiliser des fonctions définies en SpécialK dans un programme MzScheme :

Fichier `fac.spk` :

```
fac(0) = 1;
n > 0 -> fac(n) = n * fac(n - 1).
```

On peut ecrire dans un programme Scheme :

```
(require (lib "lib.ss" "specialk"))
(load-k "fac.spk")
(fac 10) ; => 3628800
```

`load-k` utilise le niveau de langage *SpécialK Light*.

5.2 Types

5.2.1 Types numériques

SpécialK reconnait les types numériques de Scheme. On peut utiliser les predicats de MzScheme sur ces types.

Entier

Une suite de un ou plusieurs chiffres eventuellement précédés du symbole '-'. Comme en Scheme, un entier est de taille variable, et peut approcher des valeurs "infinies". Exemples :

```
123
6
234276473647630000000
-120
```

Exemple au top-level :

```
> integer?(10).
true
```

Rationnel

Fraction de 2 entiers.

```
4/344
3/4
```

Exemple au top-level :

```
> rational?(1/2).
true
```

Réel

Une suite de 0 ou plusieurs chiffres, suivie d'un point, puis d'une suite de 1 ou plusieurs chiffres, le tout éventuellement précédé du symbole '-'.

```
23.0
.123
-3.34354
```

On dispose également de la notation scientifique : Une suite de 1 ou plusieurs chiffres suivis du caractère 'e' ou 'E', puis une suite de un ou plusieurs chiffres, le tout eventuellement précédé du symbole '-'. .

```
1e10
-3e-124
3E10
```

Exemple au top-level :

```
> real?(2.3443).
true
> real?(1e10).
true
```

On pourra utiliser les fonctions `floor`, `ceiling`,`round` de MzScheme pour les arrondis et les troncatures.

5.2.2 Type booléen

Les booléens s'ecrivent `true` et `false`.

```
> true.
true
> false.
false
> boolean?(true).
true
> boolean?(false).
true
```

5.2.3 Type chaîne de caractères

Une suite de caractères entre guillemets.

```
"hello"
```

Exemple au top-level :

```
> string?("hello").
true
> string-append("hello ", "world").
"hello world"
```

5.2.4 Types composites

Les doublets et les tableaux de SpécialK correspondent aux *doublets* et *vecteurs* de Scheme. Cependant, leur manipulation diffère. Ceci sera detaillé dans les sections *Doublets et listes*(5.5) et *tableaux*(5.6).

5.3 Expressions

Cette section aborde la syntaxe des expressions de SpécialK ainsi que les différents opérateurs que le langage comporte. Une expression a toujours un résultat et un type. Les opérateurs ne peuvent s'appliquer qu'à certains types. Le compilateur essaiera de vérifier au maximum le type des opérandes. Cependant, à cause du typage dynamique, cela ne sera pas toujours possible et l'utilisateur devra s'assurer des bon typages des expressions qu'il écrit. Dans le cas contraire, l'erreur ne pourra être détectée qu'à l'exécution, et le message d'erreur sera celui provoqué par l'exécution du code Scheme.

5.3.1 Variables

Un identifiant de variable peut comporter des caractères majuscules et minuscules, des chiffres, et les caractères '_', '-' et '?'. Il doit forcément commencer par une lettre. Autrement dit, cela correspond à l'expression régulière :

$$[A-Za-z]^{+}[A-Za-z\,?\_\text{-}]^{*}$$

Exemples :

```
iDentifiant
val?_06
vector-length
real?
```

Les majuscules et minuscules sont significatives dans un identifiant.

5.3.2 Opérateurs arithmétiques

Les opérateurs arithmétiques ne s'appliquent qu'a des opérandes de type numerique comme décrit dans la section 5.2.1. L'expression résultante est de type numérique. Le tableau qui suit indique le symbole, l'arité et la valeur retournée des opérateurs numeriques dans leur ordre de priorité. Les opérateurs binaires sont tous associatifs à gauche.

Les opérateurs '`div`' et '`mod`', contrairement aux autres, ne s'appliquent qu'à des opérandes de type numérique entier.

Symbole	**Arité**	**Résultat**	**Priorité**
`-`	unaire	opposé	1
`*`	binaire	multiplication	2
`/`	binaire	division	2
`div`	binaire	quotient de la division euclidienne	2
`mod`	binaire	reste de la division euclidienne	2
`+`	binaire	addition	3
`-`	binaire	soustraction	3

5.3.3 Opérateurs de comparaison

Les opérateurs de comparaison peuvent s'appliquer à différents types de SpécialK mais seuls deux éléments de même type peuvent être comparés. Les tableaux qui suivent indiquent pour chaque type les opérateurs utilisables et la comparaison effectuée. L'expression renvoie toujours un booléen. Tous les opérateurs de comparaison ont la même priorité et sont moins prioritaires que les opérateurs arithmétiques. Ils ne sont pas associatifs.

Appliqué à des opérandes de type numérique

Symbole	Arité	Résultat	Priorité
==	binaire	égalité	4
<>	binaire	différence	4
<=	binaire	inférieur ou égal	4
>=	binaire	supérieur ou égal	4
<	binaire	inférieur strict	4
>	binaire	supérieur strict	4

Appliqué à des opérandes de type booléen

Symbole	Arité	Résultat	Priorité
==	binaire	egalité	4
<>	binaire	différence	4

Appliqué à des opérandes de type doublet (liste)

Symbole	Arité	Résultat	Priorité
==	binaire	egalité (comparaison en profondeur des doublets)	4
<>	binaire	différence	4

Appliqué à des opérandes de type tableau

Symbole	Arité	Résultat	Priorité
==	binaire	egalité (comparaison des eléments du tableau)	4
<>	binaire	différence	4

5.3.4 Opérateurs booléens

Les opérateurs de type booléens peuvent s'appliquer à n'importe quel type d'opérande, étant donné qu'en SpécialK tout peut être évalué à `true` sauf `false`. L'expression résultante est de type booléen. Les opérateurs binaires booléens sont associatifs à gauche. Ils ont la même priorité et sont moins prioritaires que les opérateurs de comparaison. L'opérateur unaire ' `!` ' est aussi prioritaire que l'opérateur '`-`' unaire.

Symbole	Arité	Résultat	Priorité
`&`	binaire	*et* logique	5
`|`	binaire	*ou* logique	5
`!`	unaire	négation	1

5.3.5 Opérateur de définition

L'opérateur de definition, '`<-`', n'est utilisable que dans des expressions qui sont en dehors des définitions de fonctions. Il n'est pas associatif et doit avoir un identifiant en opérande gauche.

Exemple :

```
X <- (f(a) + 3/2) & B.
```

Cette expression a pour effet d'ajouter dans l'environnement global (3.2) l'association de l'identifiant `X` au résultat de l'expression `(f(a) + 3/2) & B`. La variable est utilisable après la ligne où elle est définie. Si une valeur était déjà associée à `X`, elle est écrasée par la nouvelle définition.

Symbole	Arité	Résultat	Priorité
`<-`	binaire	définition	7

5.3.6 Autres opérateurs

Il existe d'autres operateurs dans SpécialK. L'operateur ' `:`' de construction de doublet, et les opérateurs de manipulation de tableaux seront définis dans les sections 5.5 et 5.6.

5.3.7 Syntaxe des expressions, appel de fonction

Les opérateurs binaires décrit plus haut sont infixes. Les priorités des calculs peuvent être changées à l'aide du parenthèsage.

Un appel de fonction se fait de la manière suivante :

```
fonc_id(args)
```

où `args` est une liste d'expressions séparées par des '`,`'.

Les paramètres sont, comme en Scheme, passés par valeur. SpécialK est un langage à typage dynamique, il n'est donc pas possible de connaître à l'analyse le type de retour d'un appel de procédure ou d'une variable. N'importe quel type peut-être passé en paramètre, fonctions comprises.

L'exemple qui suit montre des expressions où l'erreur sera détectée à l'analyse :

```
a + (b & d).  /* application de l'operateur + avec une expression de type booléen */
3 < 3:nil.    /* comparaison d'une liste avec un entier */
```

L'exemple qui suit montre des expressions où l'erreur sera détecté à l'exécution si le type des variables ou le type de retour des fonctions est incorrect :

```
f(a) + b.  /* erreur si f renvoie autre chose qu'une valeur numérique */
a < b.     /* erreur si a et b ne sont pas du même type */
!b.        /* erreur si b n'est pas de type booléen */
```

5.4 Fonctions et clauses

Nous allons maintenant aborder le cœur et le point le plus particulier du langage SpécialK : la définitions de fonctions.

Une fonction en SpécialK est définie par une suite finie de clauses. À l'appel de la fonction, une des clauses qui la compose est choisie grâce à deux mécanismes que nous allons décrire dans cette section.

5.4.1 Définitions

Une fonction est une suite finie de clauses. Ces clauses ont deux formes :

– Sans garde :

partie gauche `=` *partie droite*

– Avec garde :

garde `->` *partie gauche* `=` *partie droite*

Les clauses doivent êtres séparées par le symbole '`;`' et se terminer par le symbole '`.`'.

Les symboles '`->`' et '`=`' sont aussi des séparateurs, comme l'illustre l'exemple suivant :

pg1 `=` *pd1* `;`
g1 `->` *pg2* `=` *pd2.*

Partie gauche

La partie gauche une expression syntaxiquement similaire à un appel de fonction, c'est à dire de la forme :

`nom_fonc(args)`

`nom_fonc` est l'indentifiant de la fonction que l'on est en train de définir. Toutes les clauses doivent forcément avoir le même identifiant.

`args` est une liste d'expressions séparées par le symbole '`,`'. Ces expressions peuvent être :

– une valeur littérale (chaîne, nombre, booleen).
– une variable.
– une expression utilisant l'operateur '` :`'. Tout autre opérateur est interdit.

Exemples :

```
f(x,0)
g(x,c :L,5)
```

mais pas :

```
f(x + 1)
g( !x,f(x),5 + a)
```

Garde

La garde est une expression dont le type doit être strictement booléen. Par exemple :

```
n > 0
(a & b) | c
```

mais pas :

```
n + 1
```

Partie droite

La partie droite est une expression quelconque. Par exemple :

```
n * fac(n-1)
```

On peut donc écrire la définition suivante(*suite de fibonacci*).

```
fib(0) = 1;
fib(1) = 1;
n > 1 -> fib(n) = fib(n - 1) + fib(n - 2).
```

5.4.2 Choix d'une clause

Le choix d'une clause se fait par deux mécanismes exécutés successivement :
– Le filtrage.
– L'évaluation de la garde.

Si le filtrage réussit et la garde est évaluée à *vrai* alors une clause peut être choisie. **Les clauses d'une fonctions doivent être *disjointes*, c'est à dire que l'ont pouvoir avoir un choix unique parmi les clauses.**

Le filtrage a lieu en premier. Il produira un *environnement de clause* au dessus de l'environnement global. Cet environnement de clause contient en plus la fonction en train d'être définie (indispensable pour la récursivité). La garde est alors évaluée dans cet environnement. Si elle est évaluée à *vrai* alors la clause est choisie. **Une clause sans garde**

est équivalente à une clause ayant une garde s'évaluant toujours à vrai. C'est à dire :

$$pg \texttt{ = } pd \;\Leftrightarrow\; \texttt{true -> } pg \texttt{ = } pd$$

Par la suite on ne considérera que des clauses avec garde.

Filtrage

Le *filtrage* a lieu lors de l'appel d'une fonction à l'exécution. Il représente une fonction qui à deux expressions particulières associe, s'il réussi, un environnement.

$$filtrage : E_g \times E_d \longmapsto environnement \tag{5.1}$$

Une expression de E_g peut être un variable, une valeur littérale ou une expression comportant uniquement l'opérateur ' :'. Dans cette partie, nous n'étudierons que les deux premiers cas, le troisième sera traité dans la section 5.5.

Une expression de E_d est une valeur ou une expression comportant uniquement l'opérateur ' :' et chaque élément est une valeur.

La fonction est définie de la manière suivante :

$$\begin{aligned} fitrage(c, c) &= \{\} \\ filtrage(c, d) &= \texttt{echec} \\ filtrage(X, c) &= \{X = c\} \end{aligned}$$

ou c et d sont des littéraux. X est une variable.

Les arguments de la partie gauche jouent le rôle d'expressions de E_g. Les arguments de l'appel de la procédure jouent le rôle des expressions de E_d.

La partie gauche d'une clause joue donc deux rôles.

Tout d'abord elle permet de créer un evironnement où à chaque variable de la liste des arguments sera associée à une valeur lors de l'appel de la fonction. Par exemple, le filtrage de la partie gauche :

$$f(a, b, c) \tag{5.2}$$

avec l'appel :

$$f(1, 2, 3) \tag{5.3}$$

produira l'environnement :

$$E = \{a = 1, b = 2, c = 3\} \tag{5.4}$$

Si l'argument à filtrer est une valeur littérale alors aucune variable n'est ajoutée dans l'environnement mais la partie gauche ne pourra être filtrée que si l'argument de l'appel est égal (au sens de l'opérateur ==) à celui de la partie gauche. Le filtrage joue donc, ici,

le rôle de condition pour qu'une clause puisse être choisie. On peut donc dire que si c est une valeur littérale, la clause :

$$g \rightarrow f(c) = pd \tag{5.5}$$

est équivalente à la clause :

$$g \wedge X == c \rightarrow f(X) = pd \tag{5.6}$$

où X est une variable arbitraire.

Si dans une partie gauche, la même variable apparaît plusieurs fois, alors toutes les occurences de la variable doivent être associées à la même valeur. Par exemple :

$$f(x, x) \tag{5.7}$$

est filtrable avec :

$$f(1, 1) \tag{5.8}$$

mais ne ne pas être filtré avec :

$$f(1, 2) \tag{5.9}$$

Toute clause dont le filtrage réussit devient une clause *candidate*. Si aucune clause ne peut être "filtrée" l'exécution se terminera par une erreur. On verra dans la section 5.5 que le filtrage peut aussi servir à accéder aux éléments d'une liste.

Évaluation de la garde

La garde est évaluée dans l'*environnement de clause*. Cette environnement est au-dessus de l'*environnement global*, c'est à dire que les variables que le composent écrasent celle de l'environnement global. Les gardes de toutes les clauses candidates sont évaluées. Si aucune n'est évaluée à vrai, alors l'exécution se terminera par une erreur, de même si plusieurs gardes sont évaluées à *vrai*. Pour que l'exécution se déroule normalement, il faut qu'à chaque appel de fonction une unique clause puisse être choisie.

5.4.3 Évaluation d'une clause

L'évaluation d'une clause (choisie) consiste à évaluer sa partie droite dans l'environnement engendré par le filtrage de la partie gauche. Prenons l'exemple de la factorielle :

```
fac(0) = 1;                          /* clause 1 */
n > 0 -> fac(n) = n * fac(n - 1).  /* clause 2 */
```

Voici la trace de l'appel de `fac(3).` :

– `fac(3)`
  On commence par filtrer l'appel avec les parties gauches. La partie gauche de la première clause a pour argument le littéral `0`. Elle ne peut donc être filtrée qu'avec l'appel `fac(0)`. Cette clause n'est donc pas candidate. La deuxième clause a une variable en partie gauche. Le filtrage donnera l'environnement $\{\texttt{n} = \texttt{3}\}$. Cette clause est candidate. La garde est alors évaluée à *vrai* dans cet environnement. Il n'y a pas d'autre clause candidate ; cette clause est donc choisie et sa partie droite évaluée.
– `3 * fac(3 - 1)` → `3 * fac(2)`
  Là encore la deuxieme clause est choisie.
– `3 * 2 * fac(2 - 1)` → `3 * 2 * fac(1)`

– `3 * 2 * 1 * fac(1 - 1)` → `3 * 2 * 1 * fac(0)`
  L'appel à `fac(0)` peut être filtré avec les deux clauses. C'est donc la garde de la deuxième clause qui va faire le choix. En effet, son evaluation dans l'environnement $\{\texttt{n} = \texttt{0}\}$ aura pour résultat *faux*. C'est donc la première clause qui est choisie.
– `3 * 2 * 1 * 1` → `6`

5.5 Doublets et listes

5.5.1 Définition

Un doublet de SpécialK est équivalent à un doublet de Scheme. Une liste est par conséquent soit la liste vide, soit un doublet dont le *car* est une valeur et le *cdr* une liste.

Un doublet se construit avec l'opérateur ' :'. **Cet opérateur est associatif à droite**. L'expression résultante est de type doublet. Par exemple, `1 : 2` construit le doublet (`a`, `b`).

Symbole	Arité	Résultat	Priorité
:	binaire	construction de doublet	6

La liste vide est représentée par le symbole `nil`.

Voici une suite d'expressions tapées au top-level utilisant les fonctions de MzScheme et leur résultat, montrant l'équivalence des doublets SpécialK avec les doublets MzScheme.

```
> pair?(1 :2).
true
> list?(1 : 2 : 3 : nil).
true
> null?(nil).
true
```

5.5.2 Utilisation

SpécialK est un langage qui propose une manipulation aisée des listes grâce au mécanisme du filtrage. En effet, il est possible d'écrire des expressions utilisant l'opérateur `:` dans les arguments de la partie gauche d'une clause. Par exemple, il est possible d'écrire la fonction suivante :

```
reverse(a : b) = b : a.
```

qui aura pour effet de renvoyer un doublet qui contient les éléments du doublet passé en paramètre inversés.

Le filtrage d'une expression en partie gauche, avec une expression dans un appel, peut être définie par le tableau suivant :

Partie Gauche	**Appel**	**Environnement Résultat**
`nil`	`nil`	$\{\}$
`litteral1`	`litteral1`	$\{\}$`litteral1` = `litteral2`
`x`	`expr` (variable, littéral ou expression)	$\{$`x` = `expr`$\}$
`exprg1 : exprg2`	`exprd1 : exprd2` (' :' associatif à droite)	`filtrage(exprg1, exprd2)` $\bigcup$ `filtrage(exprg2, exprd2)`

La première colonne contient l'expression qui apparaît en partie gauche, la deuxième ce qui peut être filtré avec. La troisième contient l'environnement résultant du filtrage, et montre que le filtrage se fait en profondeur. Toute situation ne se trouvant pas dans ce tableau fait échouer le filtrage et la clause ne peut être candidate.

Le tableau suivant montre diverses situations et leur résultat :

Partie Gauche	**Appel**	**Environnement Résultat**
`c1 : c2 : L`	`1 : 2 : 3 : 4 : nil`	$\{$`c1` = `1`, `c2` = `2`, `L` = `3 : 4 : nil`$\}$
`(a : b) : c`	`(1 : 2) : (3 : 4) : nil`	$\{$`a` = `1`, `b` = `2`, `c` = `(3 : 4) : nil`$\}$
`c : L`	`nil`	Échec
`a : a : L`	`1 : 2 : 3`	Échec (à cause de la double occurence de `a`)

5.5.3 Comparaisons sur les listes

Les opérateurs '`==`' et '`<>`' peuvent être utilisés sur les listes. Les opérandes sont comparées en profondeur, c'est à dire que deux listes sont égales si elles ont la même structure et contiennent les mêmes éléments. Plus simplement, on peut dire que '`==`' est équivalent au `equal?` de Scheme.

```
> 1 :2 :3 :nil == 1 :2 :3 :nil.
true.
> equal?(1 :2 :3 :nil, 1 :2 :3 :nil).
true.
```

```
> 1 :2 :3 :nil <> 1 :2 :3 :nil.
false.
>!equal?(1 :2 :3 :nil, 1 :2 :3 :nil).
false.
```

5.5.4 Applications diverses

Le filtrage peut permettre de manipuler des structures complexes sans définir des accesseurs.

Liste d'associations

La fonction suivante recherche une clé dans une liste d'association et renvoie la valeur associée à la clé ou `false` si la clé n'existe pas. On donne la forme suivante à la liste d'association :

```
(clé1 : val1) : ... : nil
```

```
search(x,nil) = false;
search(x, (x : val) : L) = val;
x <> key -> search(x, (key : val) : L) = search(x, L).
```

Arbres

On représente un arbre par le chaînage :

```
r : g : d
```

où `r` est la racine, `g` le fils gauche et `d` le fils droit.

On peut écrire la recherche dans un *arbre binaire de recherche* de cette manière :

```
search-abr(x,nil) = false;
r == x -> search-abr(x, r:g:d) = true;
x > r -> search-abr(x, r:g:d) = search-abr(x, d);
x < r -> search-abr(x, r:g:d) = search-abr(x, g).
```

5.6 Tableaux

5.6.1 Définitions

Un tableau en SpécialK peut contenir n'importe quel type d'objet. Cette structure est équivalente au *vecteurs* de Scheme. Un tableau peut être déclaré de manière littérale :

$$\{el_0,\ \ldots,\ el_n\}$$

Les el_i sont des expressions. Cette expression renvoie un pointeur sur un tableau de taille n dont les éléments sont les el_i. Les valeurs sont indicées de 0 à $n-1$. On peut également utiliser la fonction `make-tab` de la bibliothèque SpécialK.

Voici quelques exemples d'expressions tapées au top-level :

```
> {{1,2,3},"hello",4 :5 :nil}.
{{1,2,3},"hello",4 :5 :nil}.
> vector?({{1,2,3},"hello",4 :5 :nil}).
true
> tab-len({{1,2,3},"hello",4 :5 :nil}).
3.
> make-tab(5).
{0, 0 , 0 , 0, 0}
```

5.6.2 Utilisations

SpécialK dispose de 3 opérateurs sur les tableaux permettant d'effectuer les opérations suivantes :

Déréférencement

Cette opération renvoie une valeur contenue dans le tableau en temps constant. L'opérateur binaire '`[]`' s'applique à une expression.

```
expr [ index ]
```

`expr` est une expression qui doit avoir pour résultat un pointeur sur un tableau.

`index` est une expression qui doit avoir pour résultat une valeur numérique de type *entier*.

Si la valeur retournée est elle même un pointeur sur tableau on peut réappliquer l'operation.

Exemples :

```
> foo(x) = make-tab(20).
> i <- 1.
> j <- 2.
> T <- {{1,2,3},"hello",i :j :nil}.
> T.
{{1,2,3},"hello",1 :2 :nil}
> T[(i + j) div 2].
"hello"
> T[0][j].
3
```

```
> foo(10)[i].
0
```

L'opérateur '`[]`' est le plus prioritaire des opérateurs et est, de part sa syntaxe, associatif à gauche.

Symbole	Arité	Résultat	Priorité
`[]`	binaire	déréferencement	0

L'accés aux éléments d'un sous-tableau peut se faire également de la maniére suivante :

`T[i][j]` donne le même résultat que `T[i,j]`

Cependant, on verra dans le paragraphe suivant que ces deux écritures ne sont pas équivalentes.

Affectation

L'*affectation* est à distinguer de l'opération de *définition* qui ne peut être faite qu'en dehors des clauses. On peut affecter une valeur à une case indexée du tableau de la manière suivante :

```
expr [ index ] <- val
```

où `val` est une expression quelconque, `index` est une expression, ou une liste d'expressions séparées par des virgules. Le tableau pointé par `expr` se voit affecté à la case indicée par `index` la valeur `val` et ce tableau est renvoyé.

Exemple :

```
> T <- {{1,2,3},"hello",i :j :nil}.
> T[1] <- "salut".
{{1,2,3},"salut",1 :2 :nil}
```

L'opérateur '`<-`' est le moins prioritaire et il n'est pas associatif. L'expression de gauche doit être de la forme `expr[index]`.

Symbole	Arité	Résultat	Priorité
`<-`	binaire	affectation et renvoie le pointeur vers le tableau modifié	7

L'exemple qui suit montre la différence entre `T[i][j]` et `T[i,j]`.

```
> T <- {{1,2,3},"hello",i :j :nil}.
> T[0][2] <- "salut".
{1,2,"salut"}
> T <- {{1,2,3},"hello",i :j :nil}.
> T[0,2] <- "salut".
{{1,2,"salut"},"hello",1 :2 :nil}
```

En effet, dans le premier cas, on applique d'abord le déréferencement, ce qui renvoie le tableau {1,2,3} auquel on affecte la chaîne "salut" à l'index 2. Dans le deuxième cas, on affecte la chaîne "salut" à l'index 2 du sous tableau à l'index 0. On renvoie donc tout le tableau.

Permutation

La *permutation* permet d'échanger deux cases d'un tableau de la manière suivante :

```
expr [ index1 <-> index2 ]
```

Les cases indexées par index1 et index2 du tableau pointé par expr sont permutées et le pointeur sur le tableau modifié est renvoyé. La régle de priorité ne s'applique pas à cet opérateur vu qu'il ne peut apparaître qu'a un endroit bien précis.

Symbole	Arité	Résultat	Priorité
<->	binaire	permutation et renvoie le pointeur vers le tableau modifié	N/A

Exemple :

```
> T <- {{1,2,3},"hello",i :j :nil}.
> T[0,2 <-> 1].
{{1,2,"hello"},3,1 :2 :nil}
> T[0][0 <-> 1][1 <-> 2].
{2,3,1}
```

Pour finir, voici un exemple de programme utilisant un tableau : le *crible d'Ératostène*

```
/* Suppression des multiples de X dans T en partant de i.

On les fait passer à -1 */
i >= tab-len(T) -> suppr_multiples(T,x,i) = T;
i < tab-len(T) & T[i] mod x == 0
      -> suppr_multiples(T,x,i) = suppr_multiples(T[i] <- -1, x, i+1);
i < tab-len(T) & T[i] mod x <> 0
      -> suppr_multiples(T,x,i) = suppr_multiples(T, x, i+1).


/* i est initalisé à la case à partir de laquelle on commence */

i >= tab-len(T) -> crible(T,i) = T;
i < tab-len(T) & T[i] >= 0
      -> crible(T,i) = crible(suppr_multiples(T,T[i],i+1), i+1);
i < tab-len(T) & T[i] < 0 -> crible(T,i) = crible(T,i+1).
```

```
/* Construction d'un tableau d'entiers successifs */

i < tab-len(T) -> tab_entier(T,i) = tab_entier(T[i] <- i + 1, i + 1);
i >= tab-len(T) -> tab_entier(T,i) = T.


display("crible d'Ératostène").
crible(tab_entier(make-vector(40,0), 0), 1).
```

L'exécution de ce code affichera le résultat :

```
{1, 2, 3, -1, 5, -1, 7, -1, -1, -1, 11, -1, 13, -1, -1, -1, 17, -1, 19, -1,
-1, -1, 23, -1, -1, -1, -1, -1, 29, -1, 31, -1, -1, -1, -1, -1, 37, -1}
```

5.6.3 Comparaisons sur les tableaux

On peut utiliser les opérateurs '==' et '<>' sur les tableaux. Les opérandes sont comparées en profondeurs, c'est à dire que 2 tableaux sont égaux s'ils contiennents les mêmes éléments. Plus simplement, on peut dire que l'opérateur '==' appliqué aux tableaux est équivalent à l'operateur `equal?` de Scheme.

```
> {1,2,3} == {1,2,3}.
true.
> equal?({1,2,3}, {1,2,3}).
true.
> {1,2,3} <> {1,2,3}.
false.
> !equal?({1,2,3}, {1,2,3}).
false.
```

5.7 Les niveaux de langage

Ce chapitre a décrit le mode SpécialK *Classic*. Comme dit en introduction, il existe un mode *Light* qui a été conçu à cause des faibles performances du mode *Classic*.

Comme nous l'avons vu, le mode *Classic* teste systématiquement toutes les clauses pour savoir s'il y existe une unique clause sélectionnable. L'idée de départ aurait été de sélectionner une clause parmi plusieurs possible du moment que toutes les clauses produisent le même résultat. Cependant, il n'est pas possible de décider du problème de "confiance" des clauses, c'est à dire que l'on ne pouvait pas s'assurer que le programmeur

avait bien écrit des clauses soit disjointes, soit équivalentes. Le choix à donc été fait de n'autoriser que des clauses disjointes, donc de vérifier qu'une seule clause ne puisse être choisie.

Le mode *Light* teste les clauses dans l'ordre où elles sont écrites et choisit la première clause filtrable et dont la garde est évaluée à vrai. Par exemple, si on reprend la factorielle :

```
fac(0) = 1;
n > 0 -> fac(n) = n * fac(n - 1).
```

la garde de la deuxième clause est indispensable dans le cas où l'on fait l'appel `fac(0)` sinon deux clauses pourraient être sélectionnées.

Avec le mode *Light*, l'appel `fac(0)` entraîne le choix de la première clause sélectionnable dans l'ordre d'apparition. On peut donc écrire :

```
fac(0) = 1;
fac(n) = n * fac(n-1).
```

Cela a pour avantage d'accélérer l'execution, mais on ne verifie pas si les clauses sont disjointes.

Tout programme correct dans le niveau *Classic* est correct dans le niveau *Light*, mais l'inverse n'est pas vrai.

Annexe A

Tables récapitulatives

Cette partie est un récapitulatif des séparateurs, opérateurs et littéraux de SpécialK et de sa syntaxe.

A.1 Séparateurs

Symbole	Sémantique
`;`	Séparateur de clause.
`.`	Fin de définition d'une fonction.
`->`	Séparateur de garde et partie gauche.
`=`	Séparateur de partie gauche et partie droite.
`,`	Séparateur des arguments.
`( )`	Parenthèsage des expressions, appel de procédure.
`{ }`	Début et fin de création d'un tableau.

A.2 Opérateurs

Symbole	Arité	Associativité	Sémantique	Priorité
`[...]`	binaire	gauche	déréférencement	0
`!`	unaire	N/A	négation	1
`-`	unaire	N/A	opposé	1
`*`	binaire	gauche	multiplication	2
`/`	binaire	gauche	division	2
`div`	binaire	gauche	quotient de la division euclidienne	2
`mod`	binaire	gauche	reste de la division euclidienne	2
`+`	binaire	gauche	addition	3
`-`	binaire	gauche	soustraction	3
`==`	binaire	non	égalité	4
`<>`	binaire	non	différence	4
`<=`	binaire	non	inférieur ou égale	4
`>=`	binaire	non	supérieur ou égale	4
`<`	binaire	non	inférieur strict	4
`>`	binaire	non	supérieur strict	4
`&`	binaire	gauche	*et* logique	5
`|`	binaire	gauche	*ou* logique	5
`:`	binaire	droite	construction de doublet	6
`<-`	binaire	non	affectation et renvoie le pointeur vers le tableau modifié	7
`<->`	binaire	non	permutation et renvoie le pointeur vers le tableau modifié	N/A

A.3 Littéraux

Représentation	Sémantique
`nil`	Liste vide
`[0-9]+`	Entier
`[0-9]*.[0-9]+`	Réel
`[A-Za-z_][A-Za-z_0-9?]*`	Identifiant
`true`,`false`	Booléen
`".*"`	Chaîne de caractères

Annexe B

Grammaire

L'analyseur a été fait à l'aide des outils de type *lex&yacc* des *parser tools* de PLT Scheme. La grammaire telle quelle est fortement ambiguë. Ces ambiguités ont été levées en utilisant les systèmes offerts par *yacc*.

Le symbole υ représente la chaîne vide.

prog	→	line prog |
		υ
line	→	list-clause `.` |
		expr-t `.`
list-clause	→	clause `;` list-clause |
		clause
clause	→	expr `->` partie-gauche `=` expr |
		partie-gauche `=` expr
partie-gauche	→	ID `(` list-expr `)` |
		ID `(` `)`
expr	→	expr `+` expr |
		expr `*` expr |
		expr `-` expr |
		expr `/` expr |
		expr `div` expr |
		expr `mod` expr |
		expr `==` expr |
		expr `<>` expr |
		expr `<` expr |
		expr `<=` expr |
		expr `>` expr |
		expr `>=` expr |
		expr `&` expr |
		expr `|` expr |
		expr `:` expr |

		`(` expr `)` |
		ID `(` list-expr `)` |
		ID `(` `)` |
		`!` expr |
		`-` expr |
		`{` `}` |
		`{` list-expr `}` |
		expr `[` list-expr `]` |
		expr `[` list-expr `<->` list-expr `]` |
		expr `[` list-expr `]` `<-` expr |
		INTEGER |
		REAL |
		ID |
		STRING |
		`nil`
expr-t	→	expr |
		ID `<-` expr
list-expr	→	expr `,` list-expr |
		expr

Annexe C

GNU Free Documentation License

Preamble

The purpose of this License is to make a manual, textbook, or other functional and useful document "free" in the sense of freedom : to assure everyone the effective freedom to copy and redistribute it, with or without modifying it, either commercially or noncommercially. Secondarily, this License preserves for the author and publisher a way to get credit for their work, while not being considered responsible for modifications made by others.

This License is a kind of "copyleft", which means that derivative works of the document must themselves be free in the same sense. It complements the GNU General Public License, which is a copyleft license designed for free software.

We have designed this License in order to use it for manuals for free software, because free software needs free documentation : a free program should come with manuals providing the same freedoms that the software does. But this License is not limited to software manuals ; it can be used for any textual work, regardless of subject matter or whether it is published as a printed book. We recommend this License principally for works whose purpose is instruction or reference.

1. APPLICABILITY AND DEFINITIONS

This License applies to any manual or other work, in any medium, that contains a notice placed by the copyright holder saying it can be distributed under the terms of this License. Such a notice grants a world-wide, royalty-free license, unlimited in duration, to use that work under the conditions stated herein. The **"Document"**, below, refers to any such

manual or work. Any member of the public is a licensee, and is addressed as **"you"**. You accept the license if you copy, modify or distribute the work in a way requiring permission under copyright law.

A **"Modified Version"** of the Document means any work containing the Document or a portion of it, either copied verbatim, or with modifications and/or translated into another language.

A **"Secondary Section"** is a named appendix or a front-matter section of the Document that deals exclusively with the relationship of the publishers or authors of the Document to the Document's overall subject (or to related matters) and contains nothing that could fall directly within that overall subject. (Thus, if the Document is in part a textbook of mathematics, a Secondary Section may not explain any mathematics.) The relationship could be a matter of historical connection with the subject or with related matters, or of legal, commercial, philosophical, ethical or political position regarding them.

The **"Invariant Sections"** are certain Secondary Sections whose titles are designated, as being those of Invariant Sections, in the notice that says that the Document is released under this License. If a section does not fit the above definition of Secondary then it is not allowed to be designated as Invariant. The Document may contain zero Invariant Sections. If the Document does not identify any Invariant Sections then there are none.

The **"Cover Texts"** are certain short passages of text that are listed, as Front-Cover Texts or Back-Cover Texts, in the notice that says that the Document is released under this License. A Front-Cover Text may be at most 5 words, and a Back-Cover Text may be at most 25 words.

A **"Transparent"** copy of the Document means a machine-readable copy, represented in a format whose specification is available to the general public, that is suitable for revising the document straightforwardly with generic text editors or (for images composed of pixels) generic paint programs or (for drawings) some widely available drawing editor, and that is suitable for input to text formatters or for automatic translation to a variety of formats suitable for input to text formatters. A copy made in an otherwise Transparent file format whose markup, or absence of markup, has been arranged to thwart or discourage subsequent modification by readers is not Transparent. An image format is not Transparent if used for any substantial amount of text. A copy that is not "Transparent" is called **"Opaque"**.

Examples of suitable formats for Transparent copies include plain ASCII without markup, Texinfo input format, LaTeX input format, SGML or XML using a publicly available DTD, and standard-conforming simple HTML, PostScript or PDF designed for human modification. Examples of transparent image formats include PNG, XCF and JPG. Opaque formats include proprietary formats that can be read and edited only by proprietary word processors, SGML or XML for which the DTD and/or processing tools are not generally available, and the machine-generated HTML, PostScript or PDF produced by some word processors for output purposes only.

The **"Title Page"** means, for a printed book, the title page itself, plus such following pages as are needed to hold, legibly, the material this License requires to appear in the title page. For works in formats which do not have any title page as such, "Title Page" means the text near the most prominent appearance of the work's title, preceding the beginning

of the body of the text.

A section **"Entitled XYZ"** means a named subunit of the Document whose title either is precisely XYZ or contains XYZ in parentheses following text that translates XYZ in another language. (Here XYZ stands for a specific section name mentioned below, such as **"Acknowledgements"**, **"Dedications"**, **"Endorsements"**, or **"History"**.) To **"Preserve the Title"** of such a section when you modify the Document means that it remains a section "Entitled XYZ" according to this definition.

The Document may include Warranty Disclaimers next to the notice which states that this License applies to the Document. These Warranty Disclaimers are considered to be included by reference in this License, but only as regards disclaiming warranties : any other implication that these Warranty Disclaimers may have is void and has no effect on the meaning of this License.

2. VERBATIM COPYING

You may copy and distribute the Document in any medium, either commercially or noncommercially, provided that this License, the copyright notices, and the license notice saying this License applies to the Document are reproduced in all copies, and that you add no other conditions whatsoever to those of this License. You may not use technical measures to obstruct or control the reading or further copying of the copies you make or distribute. However, you may accept compensation in exchange for copies. If you distribute a large enough number of copies you must also follow the conditions in section 3.

You may also lend copies, under the same conditions stated above, and you may publicly display copies.

3. COPYING IN QUANTITY

If you publish printed copies (or copies in media that commonly have printed covers) of the Document, numbering more than 100, and the Document's license notice requires Cover Texts, you must enclose the copies in covers that carry, clearly and legibly, all these Cover Texts : Front-Cover Texts on the front cover, and Back-Cover Texts on the back cover. Both covers must also clearly and legibly identify you as the publisher of these copies. The front cover must present the full title with all words of the title equally prominent and visible. You may add other material on the covers in addition. Copying with changes limited to the covers, as long as they preserve the title of the Document and satisfy these conditions, can be treated as verbatim copying in other respects.

If the required texts for either cover are too voluminous to fit legibly, you should put the first ones listed (as many as fit reasonably) on the actual cover, and continue the rest onto adjacent pages.

If you publish or distribute Opaque copies of the Document numbering more than 100, you must either include a machine-readable Transparent copy along with each Opaque copy, or state in or with each Opaque copy a computer-network location from which the general network-using public has access to download using public-standard network protocols a

complete Transparent copy of the Document, free of added material. If you use the latter option, you must take reasonably prudent steps, when you begin distribution of Opaque copies in quantity, to ensure that this Transparent copy will remain thus accessible at the stated location until at least one year after the last time you distribute an Opaque copy (directly or through your agents or retailers) of that edition to the public.

It is requested, but not required, that you contact the authors of the Document well before redistributing any large number of copies, to give them a chance to provide you with an updated version of the Document.

4. MODIFICATIONS

You may copy and distribute a Modified Version of the Document under the conditions of sections 2 and 3 above, provided that you release the Modified Version under precisely this License, with the Modified Version filling the role of the Document, thus licensing distribution and modification of the Modified Version to whoever possesses a copy of it. In addition, you must do these things in the Modified Version :

A. Use in the Title Page (and on the covers, if any) a title distinct from that of the Document, and from those of previous versions (which should, if there were any, be listed in the History section of the Document). You may use the same title as a previous version if the original publisher of that version gives permission.

B. List on the Title Page, as authors, one or more persons or entities responsible for authorship of the modifications in the Modified Version, together with at least five of the principal authors of the Document (all of its principal authors, if it has fewer than five), unless they release you from this requirement.

C. State on the Title page the name of the publisher of the Modified Version, as the publisher.

D. Preserve all the copyright notices of the Document.

E. Add an appropriate copyright notice for your modifications adjacent to the other copyright notices.

F. Include, immediately after the copyright notices, a license notice giving the public permission to use the Modified Version under the terms of this License, in the form shown in the Addendum below.

G. Preserve in that license notice the full lists of Invariant Sections and required Cover Texts given in the Document's license notice.

H. Include an unaltered copy of this License.

I. Preserve the section Entitled "History", Preserve its Title, and add to it an item stating at least the title, year, new authors, and publisher of the Modified Version as given on the Title Page. If there is no section Entitled "History" in the Document, create one stating the title, year, authors, and publisher of the Document as given on its Title Page, then add an item describing the Modified Version as stated in the previous sentence.

J. Preserve the network location, if any, given in the Document for public access to a Transparent copy of the Document, and likewise the network locations given in

the Document for previous versions it was based on. These may be placed in the "History" section. You may omit a network location for a work that was published at least four years before the Document itself, or if the original publisher of the version it refers to gives permission.

K. For any section Entitled "Acknowledgements" or "Dedications", Preserve the Title of the section, and preserve in the section all the substance and tone of each of the contributor acknowledgements and/or dedications given therein.

L. Preserve all the Invariant Sections of the Document, unaltered in their text and in their titles. Section numbers or the equivalent are not considered part of the section titles.

M. Delete any section Entitled "Endorsements". Such a section may not be included in the Modified Version.

N. Do not retitle any existing section to be Entitled "Endorsements" or to conflict in title with any Invariant Section.

O. Preserve any Warranty Disclaimers.

If the Modified Version includes new front-matter sections or appendices that qualify as Secondary Sections and contain no material copied from the Document, you may at your option designate some or all of these sections as invariant. To do this, add their titles to the list of Invariant Sections in the Modified Version's license notice. These titles must be distinct from any other section titles.

You may add a section Entitled "Endorsements", provided it contains nothing but endorsements of your Modified Version by various parties–for example, statements of peer review or that the text has been approved by an organization as the authoritative definition of a standard.

You may add a passage of up to five words as a Front-Cover Text, and a passage of up to 25 words as a Back-Cover Text, to the end of the list of Cover Texts in the Modified Version. Only one passage of Front-Cover Text and one of Back-Cover Text may be added by (or through arrangements made by) any one entity. If the Document already includes a cover text for the same cover, previously added by you or by arrangement made by the same entity you are acting on behalf of, you may not add another; but you may replace the old one, on explicit permission from the previous publisher that added the old one.

The author(s) and publisher(s) of the Document do not by this License give permission to use their names for publicity for or to assert or imply endorsement of any Modified Version.

5. COMBINING DOCUMENTS

You may combine the Document with other documents released under this License, under the terms defined in section 4 above for modified versions, provided that you include in the combination all of the Invariant Sections of all of the original documents, unmodified, and list them all as Invariant Sections of your combined work in its license notice, and that you preserve all their Warranty Disclaimers.

The combined work need only contain one copy of this License, and multiple identical Invariant Sections may be replaced with a single copy. If there are multiple Invariant

Sections with the same name but different contents, make the title of each such section unique by adding at the end of it, in parentheses, the name of the original author or publisher of that section if known, or else a unique number. Make the same adjustment to the section titles in the list of Invariant Sections in the license notice of the combined work.

In the combination, you must combine any sections Entitled "History" in the various original documents, forming one section Entitled "History" ; likewise combine any sections Entitled "Acknowledgements", and any sections Entitled "Dedications". You must delete all sections Entitled "Endorsements".

6. COLLECTIONS OF DOCUMENTS

You may make a collection consisting of the Document and other documents released under this License, and replace the individual copies of this License in the various documents with a single copy that is included in the collection, provided that you follow the rules of this License for verbatim copying of each of the documents in all other respects.

You may extract a single document from such a collection, and distribute it individually under this License, provided you insert a copy of this License into the extracted document, and follow this License in all other respects regarding verbatim copying of that document.

7. AGGREGATION WITH INDEPENDENT WORKS

A compilation of the Document or its derivatives with other separate and independent documents or works, in or on a volume of a storage or distribution medium, is called an "aggregate" if the copyright resulting from the compilation is not used to limit the legal rights of the compilation's users beyond what the individual works permit. When the Document is included in an aggregate, this License does not apply to the other works in the aggregate which are not themselves derivative works of the Document.

If the Cover Text requirement of section 3 is applicable to these copies of the Document, then if the Document is less than one half of the entire aggregate, the Document's Cover Texts may be placed on covers that bracket the Document within the aggregate, or the electronic equivalent of covers if the Document is in electronic form. Otherwise they must appear on printed covers that bracket the whole aggregate.

8. TRANSLATION

Translation is considered a kind of modification, so you may distribute translations of the Document under the terms of section 4. Replacing Invariant Sections with translations requires special permission from their copyright holders, but you may include translations of some or all Invariant Sections in addition to the original versions of these Invariant Sections. You may include a translation of this License, and all the license notices in the Document, and any Warranty Disclaimers, provided that you also include the original English version of this License and the original versions of those notices and disclaimers.

In case of a disagreement between the translation and the original version of this License or a notice or disclaimer, the original version will prevail.

If a section in the Document is Entitled "Acknowledgements", "Dedications", or "History", the requirement (section 4) to Preserve its Title (section 1) will typically require changing the actual title.

9. TERMINATION

You may not copy, modify, sublicense, or distribute the Document except as expressly provided for under this License. Any other attempt to copy, modify, sublicense or distribute the Document is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.

10. FUTURE REVISIONS OF THIS LICENSE

The Free Software Foundation may publish new, revised versions of the GNU Free Documentation License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns. See http ://www.gnu.org/copyleft/.

Each version of the License is given a distinguishing version number. If the Document specifies that a particular numbered version of this License "or any later version" applies to it, you have the option of following the terms and conditions either of that specified version or of any later version that has been published (not as a draft) by the Free Software Foundation. If the Document does not specify a version number of this License, you may choose any version ever published (not as a draft) by the Free Software Foundation.

ADDENDUM : How to use this License for your documents

To use this License in a document you have written, include a copy of the License in the document and put the following copyright and license notices just after the title page :

> Copyright ©YEAR YOUR NAME. Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.2 or any later version published by the Free Software Foundation ; with no Invariant Sections, no Front-Cover Texts, and no Back-Cover Texts. A copy of the license is included in the section entitled "GNU Free Documentation License".

If you have Invariant Sections, Front-Cover Texts and Back-Cover Texts, replace the "with...Texts." line with this :

with the Invariant Sections being LIST THEIR TITLES, with the Front-Cover Texts being LIST, and with the Back-Cover Texts being LIST.

If you have Invariant Sections without Cover Texts, or some other combination of the three, merge those two alternatives to suit the situation.

If your document contains nontrivial examples of program code, we recommend releasing these examples in parallel under your choice of free software license, such as the GNU General Public License, to permit their use in free software.

Index