Ce document ne décrit que le sous-ensemble de Scheme enseigné dans l'UE Programmation récursive[1] de l'UPMC. Cette carte de référence peut être librement utilisée lors des contrôles de connaissance.

Spécification, signature, définition

```
;;; <nom-fonction>: <type-fonction>                ←signature
;;; (<nom-fonction> <variable>*) sémantique...
;;; ERREUR lorsque ...
;;; HYPOTHÈSE: ...
(define (<nom-fonction> <variable>*)               ←définition
  ... )
```

Grammaire des types

<AutreType> → *Tout type dont l'usage est préconisé par un exercice.*

<types> → *<type>*
<type> `*` *<types>*

<type> → *<type-non-contraint>*
<type-contraint>

<type-destination> → *<type>*
`Rien`

<type-non-contraint> → *<type-base>*
`LISTE` [*<type>*]
`COUPLE` [*<type>* *<type>*]
`VECTEUR` [*<type>*]
`NUPLET` [*<type>*`*`]
<type> `^` *<nat>*
<type> `+ #f`
(*<type-fonction>*)

<type-base> → `nat` *ou* `int` *ou* `float` *ou* `Nombre` *ou* `string` *ou* `bool` *ou* `Valeur` *ou* `Image` *ou* `symbol` *ou* *<variable-type>* *ou* *<AutreType>*

<type-contraint> → *<type>* `/` *<contrainte>* `/`

<type-fonction> → *<types>* `->` *<type-destination>*
`->` *<type-destination>*

<variable-type> → `alpha`
`beta`
etc.

Grammaire du langage

<programme> → *<expression-ou-définition>*`*`

<alternant> → *<expression>*

<alternative> → (`if` *<condition>* *<conséquence>* *<alternant>*)

<application> → (*<fonction>* *<argument>*`*`)

<argument> → *<expression>*

<bloc> → (`let` (*<liaison>*`*`) *<corps>*)
(`let *` (*<liaison>*`*`) *<corps>*)

<booléen> → `#t`
`#f`

CHAÎNE → `"` *caractères autres que guillemets* `"`

<citation> → (`quote` *<S-expression>*)
`'` *<S-expression>*

<clause> → (*<condition>* *<expression>*)

<clauses> → *<clause>* *<clause>*`*`
<clause>`*`(`else` *<expression>*)

<condition> → *<expression>*

<conditionnelle> → (`cond` *<clauses>*)

<conjonction> → (`and` *<expression>*`*`)

<conjonction-ou-disjonction> → *<conjonction>*
<disjonction>

<conséquence> → *<expression>*

<constante> → *<booléen>*
NOMBRE
CHAÎNE

<corps> → *<définition>*`*` *<expression>*

<définition> → (`define` (*<nom-fonction>* *<variable>*`*`) *<corps>*

<disjonction> → (`or` *<expression>*`*`)

<expression> → *<constante>*
<variable>
<nom-fonction>
<forme-spéciale>
<application>

<expression-ou-définition> → *<expression>*
<définition>

<fonction> → *<expression>*

<forme-spéciale> → *<conditionnelle>*
<alternative>
<conjonction-ou-disjonction>
<citation>
<bloc>

IDENTIFICATEUR → *tous les symboles de Scheme qui ne sont pas des mots-clés c'est-à-dire* `cond` , `else` , `if` , `quote` , `begin` , `let` , `let *`, `define` , `or` , `and` .

<liaison> → (*<variable>* *<expression>*)

NOMBRE → *tous les nombres de Scheme*

<nom-fonction> → IDENTIFICATEUR

<S-expression> → *<constante>*
SYMBOLE
<vecteur>
(*<S-expression>*`*`)

SYMBOLE → *tous les symboles de Scheme*

<variable> → IDENTIFICATEUR

<vecteur> → `#(` *<S-expression>*`*`)

Commentaires

```
;;; Commentaire général
    ;; Commentaire de bloc
                    ; petit commentaire local
```

Un commentaire général porte sur les expressions ou définitions qui suivent. Il débute toujours en première colonne.

Un commentaire de bloc qualifie l'expression ou la définition qui suit. Il est aligné avec cette expression ou définition.

Un commentaire local porte sur l'expression de la même ligne qui précéde immédiatement ce commentaire. Il s'achève en bout de ligne.

Bibliothèque

Les nombres peuvent être des entiers, des rationnels ou des flottants.

`equal?`: **Valeur** × **Valeur** → **bool**
(`equal?` v_1 v_2) *compare deux valeurs quelconques*

`not`: **bool** → **bool**
(`not` b) *rend la négation de b*

`number?`: **Valeur** → **bool**
(`number?` v) *reconnaît si v est un nombre*

`integer?`: **Valeur** → **bool**
(`integer?` v) *reconnaît si v est entier*

`positive?`: **Nombre** → **bool**
(`positive?` x) *vérifie que x est strictement positif*

`negative?`: **Nombre** → **bool**
(`negative?` x) *vérifie que x est strictement négatif*

`odd?`: **int** → **bool**
(`odd?` n) *vérifie que n est impair*

`even?`: **int** → **bool**
(`even?` n) *vérifie que n est pair*

`=`: **Nombre** × **Nombre** → **bool**
(`=` x_1 x_2) *compare deux nombres*

`+`: **Nombre** × **Nombre** × ... → **Nombre**
(`+` x`...`) *rend la somme de ses arguments*

`-`: **Nombre** × **Nombre** → **Nombre**
(`-` x_1 x_2) *rend* $x_1 - x_2$

`*`: **Nombre** × **Nombre** × ... → **Nombre**
(`*` x`...`) *rend le produit de ses arguments*

`/`: **Nombre** × **Nombre**/$\neq 0$/ → **Nombre**
(`/` x_1 x_2) *rend* x_1/x_2

`>`: **Nombre** × **Nombre** → **bool**
(`>` x_1 x_2) *vérifie que* $x_1 > x_2$
Prédicats analogues: `>=` , `<` *et* `<=`

`quotient`: **int** × **int**/$\neq 0$/ → **int**
(`quotient` n_1 n_2) *rend le quotient de la division euclidienne de* n_1 *par* n_2

`remainder`: **int** × **int**/$\neq 0$/ → **int**
(`remainder` n_1 n_2) *rend le reste de la division euclidienne de* n_1 *par* n_2

`modulo`: **int** × **int**/$\neq 0$/ → **int**
(`modulo` n_1 n_2) *rend le reste de la division euclidienne de* n_1 *par* n_2

`max`: **Nombre** × **Nombre** × ... → **Nombre**
(`max` x`...`) *rend le plus grand des x*

`min`: **Nombre** × **Nombre** × ... → **Nombre**
(`min` x`...`) *rend le plus petit des x*

`abs`: **Nombre** → **Nombre**
(`abs` x) *rend la valeur absolue de x*

`sqrt`: **Nombre**/≥ 0/ → **Nombre**
(`sqrt` x) *rend la racine carrée de x*

`pair?`: **Valeur** → **bool**
(`pair?` v) *vérifie que v a un* `car` *et un* `cdr`

`list?`: **Valeur** → **bool**
(`list?` v) *reconnaît que v est une liste (éventuellement vide)*

`car`: **LISTE**[α]/`pair?`/ → α
(`car` ℓ) *rend le premier terme de la liste* ℓ. *[*`ERREUR` *lorsque* ℓ *ne satisfait pas* `pair?` *]*

`cdr`: **LISTE**[α]/`pair?`/ → **LISTE**[α]
(`cdr` ℓ) *rend la liste formée de tous les termes de* ℓ *sauf son premier. [*`ERREUR` *lorsque* ℓ *ne satisfait pas* `pair?` *]*
(`cadr` ℓ) ≡ (`car` (`cdr` ℓ))
(`cddr` ℓ) ≡ (`cdr` (`cdr` ℓ)) etc.

`cons`: α × **LISTE**[α] → **LISTE**[α]
(`cons` v ℓ) *crée une nouvelle liste telle que son* `car` *est v et son* `cdr` *est* ℓ

`list`: α × ... → **LISTE**[α]
(`list` v`...`) *crée une liste dont les termes sont les arguments.*
(`list`) *rend la liste vide*

[1] http://www-licence.ufr-info-p6.jussieu.fr/lmd/licence/2005/ue/prec-2005oct/

`append:` **LISTE**$[\alpha] \times \ldots \rightarrow$ **LISTE**$[\alpha]$
`(append` ℓ`...)` *rend la concaténation des listes reçues en arguments*

`reverse:` **LISTE**$[\alpha] \rightarrow$ **LISTE**$[\alpha]$
`(reverse` ℓ`)` *rend la liste renversée de* ℓ

`member:` $\alpha \times$ **LISTE**$[\alpha] \rightarrow$ **LISTE**$[\alpha] +$ `#f`
`(member` v ℓ`)` *rend le suffixe de* ℓ *débutant par v ou #f si v n'apparaît pas dans* ℓ

`assoc:` $\alpha \times$ **LISTE**[**COUPLE**$[\alpha\ \beta]] \rightarrow$ **COUPLE**$[\alpha\ \beta] +$ `#f`
`(assoc` c al`)` *rend le couple (*clé valeur*) dont la clé est c ou bien #f s'il n'y a pas de tel couple*

`symbol?:` **Valeur** → **bool**
`(symbol?` v`)` *reconnaît que v est un symbole*

`string?:` **Valeur** → **bool**
`(string?` v`)` *reconnaît que v est une chaîne de caractères*

`string-length:` **string** → **nat**
`(string-length` s`)` *rend la longueur de la chaîne s*

`substring:` **string** × **nat** × **nat** → **string**
`(substring` s i j`)` *construit la chaîne à partir des caractères* $[i \ldots j[$ *de s*

`string-append:` **string** × ... → **string**
`(string-append` s`...)` *construit la concaténation de toutes les chaînes reçues en arguments*

`vector?:` **Valeur** → **bool**
`(vector?` v`)` *reconnaît que v est un vecteur*

`vector:` $\alpha \times \ldots \rightarrow$ **VECTEUR**$[\alpha]$
`(vector` v`...)` *rend le vecteur dont les termes sont les arguments*

`vector-ref:` **VECTEUR**$[\alpha] \rightarrow \alpha$
`(vector-ref` *vecteur index*`)` *rend le index-ième terme du vecteur*

`vector-length:` **VECTEUR**$[\alpha] \rightarrow$ **nat**
`(vector-length` *vecteur*`)` *rend la longueur du vecteur*

`vector->list:` **VECTEUR**$[\alpha] \rightarrow$ **LISTE**$[\alpha]$
`(vector->list` *vecteur*`)` *rend la liste des termes du vecteur*

`list->vector:` **LISTE**$[\alpha] \rightarrow$ **VECTEUR**$[\alpha]$
`(list->vector` *liste*`)` *rend le vecteur des termes de la liste*

`map:` $(\alpha \rightarrow \beta) \times$ **LISTE**$[\alpha] \rightarrow$ **LISTE**$[\beta]$
`(map` f ℓ`)` *rend la liste dont les termes sont les images par f des termes de la liste* ℓ

`display:` **Valeur** → **Rien**
`(display` v`)` *imprime la valeur, ne rend rien d'intéressant*

`newline:` → **Rien**
`(newline)` *passe à la ligne, ne rend rien d'intéressant*

Suppléments

Les fonctions et formes spéciales suivantes ne sont utilisables sous DrScheme que si le paxon (ou *teachpack*) LI101 est actif.

`filtre:` $(\alpha \rightarrow$ **bool**$) \times$ **LISTE**$[\alpha] \rightarrow$ **LISTE**$[\alpha]$
`(filtre` p ℓ`)` *rend la liste extraite de* ℓ *dont les termes satisfont le prédicat p*

`reduce:` $(\alpha \times \beta \rightarrow \beta) \times \beta \times$ **LISTE**$[\alpha] \rightarrow \beta$
`(reduce` f e ℓ`)` *Compose binairement par f les éléments de la liste* ℓ *en terminant par e. Plus formellement:* `(reduce f e (list` e_1 e_2 ... e_n`))` $\equiv$ `(`f e_1 `(`f e_2 `...(`f e_n e`) ...))`.

`erreur:` **symbol** $\times \alpha \times \ldots \rightarrow$ 🐞
`(erreur` f`...)` *signale une erreur avec f comme nom de fonction et les arguments suivants comme explication de l'erreur. Signaler une erreur stoppe l'évaluation.*

`erreur?:` $(\alpha \times \ldots \rightarrow \beta) \times \alpha \times \ldots \rightarrow$ **bool**
`(erreur?` f`...)` *teste si une fonction f appliquée à ses arguments provoque une erreur.*

`verifier:` $\ldots \rightarrow$ **bool**
`(verifier` *nom sexp* = *sexp*`...)` *regroupe les tests relatifs à la fonction* nom. *Évalue tour à tour chaque S-expression avant et après le signe d'équivalence* = *et vérifie que les valeurs sont égales. Rend* `#t` *si tous les tests sont corrects.*

Bibliothèque graphique

Le système de coordonnées des primitives graphiques varie de $(-1,-1)$ pour le coin bas gauche à $(+1,+1)$ pour le coin haut droit. Le type **Coordonnée** désigne un nombre entre -1 et $+1$. Les images construites sont par défaut carrées et de côté 100 pixels.

`image-vide:` → **Image**
`(image-vide)` *rend une image carrée blanche.*

`filled-triangle:` **Coordonnée**[6] → **Image**
`(filled-triangle` x_0 y_0 x_1 y_1 x_2 y_2`)` *rend une image carrée blanche contenant un triangle noir de sommets* (x_0,y_0), (x_1,y_1) *et* (x_2,y_2).

`invert:` **Image** → **Image**
`(invert` *image*`)` *rend une nouvelle image à couleur inversée.*

`line:` **Coordonnée**[4] → **Image**
`(line` x_0 y_0 x_1 y_1`)` *rend une image carrée blanche contenant un segment noir partant de* (x_0,y_0) *jusqu'à* (x_1,y_1).

`overlay:` **Image** × ... → **Image**
`(overlay` *image*`...)` *rend une image superposant les contenus des images reçues en arguments [*`ERREUR` *lorsque les images sont de tailles différentes]*

`quarter-turn-right:` **Image** → **Image**
`(quarter-turn-right` *image*`)` *rend une nouvelle image dont le contenu est celui de l'*image *tournée de 90 degrés dans le sens des aiguilles d'une montre.*

`stack:` **Image** × **Image** → **Image**
`(stack` $image_1$ $image_2$`)` *rend une nouvelle image par juxtaposition de l'*image_1 *au dessus de l'*image2. *[*`ERREUR` *lorsque les images sont de largeurs différentes]*

`image?:` **Valeur** → **bool**
`(image?` v`)` *reconnaît si v est une image*

`image-width:` **Image** → **nat**
`(image-width` *image*`)` *rend la largeur d'une image (en pixels).*

`image-height:` **Image** → **nat**
`(image-height` *image*`)` *rend la hauteur d'une image (en pixels).*

`resize-image:` **Image** × **nat** × **nat** → **Image**
`(resize-image` *image largeur hauteur*`)` *rend une nouvelle image homothétique avec une largeur et une hauteur spécifiées (en pixels).*

`mirror-image:` **Image** → **Image**
`(mirror-image` *image*`)` *rend une image miroir (symétrie y) de l'image fournie.*

Lignes et paragraphes

Une ligne est une chaîne de caractères sans fin de ligne. Un paragraphe est une chaîne de caractères débutant par une fin de ligne suivie de lignes, chacune terminée par une fin de ligne. Par exemple, `"bon jour"` est une ligne, tandis que la chaîne suivante est un paragraphe formé de deux lignes: `"`
`bon`
`jour`
`"`

`paragraphe:` **LISTE**[**Ligne**] → **Paragraphe**
`(paragraphe` *lignes*`)` *rend le paragraphe formé des lignes*

`paragraphe-cons:` **Ligne** × **Paragraphe** → **Paragraphe**
`(paragraphe-cons` *ligne paragraphe*`)` *construit un paragraphe ayant en premier ligne suivi des lignes de paragraphe*

`lignes:` **Paragraphe** → **LISTE**[**Ligne**]
`(lignes` *paragraphe*`)` *rend la liste des lignes que comporte le paragraphe*

`paragraphe-append:` **Paragraphe** × ... → **Paragraphe**
`(paragraphe-append` *paragraphes*`...)` *rend le paragraphe correspondant à la concaténation des paragraphes reçus en arguments*

`->string:` **Valeur** → **string**
`(->string` v`)` *construit une chaîne représentant l'argument qui peut être quelconque (nombre, symbole, chaîne, liste, etc.)*

Bibliothèque d'arbres

Les fonctions préfixées par `ab-` forment la barrière d'abstraction des arbres binaires. Les fonctions préfixées par `ag-` forment la barrière d'abstraction des arbres généraux.

La barrière d'abstraction est formée de constructeur(s), sélecteurs (ou accesseurs) et reconnaisseur(s).

`ab-noeud:` $\alpha \times$ **ArbreBinaire**$[\alpha] \times$ **ArbreBinaire**$[\alpha] \rightarrow$ **ArbreBinaire**$[\alpha]$
`(ab-noeud` *etiquette arbre−gauche arbre−droit*`)` *construit un arbre binaire*

`ab-vide:` → **ArbreBinaire**$[\alpha]$
`(ab-vide)` *construit un arbre binaire vide*

`ab-noeud?:` **ArbreBinaire**$[\alpha] \rightarrow$ **bool**
`(ab-noeud?` *arbre*`)` *reconnaît si un arbre binaire n'est pas vide*

`ab-etiquette:` **ArbreBinaire**$[\alpha] \rightarrow \alpha$
`(ab-etiquette` *arbre*`)` *rend l'étiquette d'un arbre binaire non vide*

`ab-gauche:` **ArbreBinaire**$[\alpha] \rightarrow$ **ArbreBinaire**$[\alpha]$
`(ab-gauche` *arbre*`)` *rend le sous-arbre gauche d'un arbre binaire non vide*

`ab-droit:` **ArbreBinaire**$[\alpha] \rightarrow$ **ArbreBinaire**$[\alpha]$
`(ab-droit` *arbre*`)` *rend le sous-arbre droit d'un arbre binaire non vide*

`ab-expression:` **ArbreBinaire**$[\alpha] \rightarrow$ **Expression**
`(ab-expression` *arbre*`)` *rend l'expression construisant l'arbre binaire*

`ag-noeud:` $\alpha \times$ **LISTE**[**ArbreGeneral**$[\alpha]] \rightarrow$ **ArbreGeneral**$[\alpha]$
`(ag-noeud` *etiquette sous−arbres*`)` *construit un arbre général*

`ag-etiquette:` **ArbreGeneral**$[\alpha] \rightarrow \alpha$
`(ag-etiquette` *arbre*`)` *rend l'étiquette d'un arbre général*

`ag-foret:` **ArbreGeneral**$[\alpha] \rightarrow$ **LISTE**[**ArbreGeneral**$[\alpha]]$
`(ag-foret` *arbre*`)` *rend la forêt sous la racine de l'arbre*

`ag-expression:` **ArbreGeneral**$[\alpha] \rightarrow$ **Expression**
`(ag-expression` *arbre*`)` *rend l'expression construisant l'arbre général*