

ment ne décrit que le sous-ensemble de Scheme enseigné dans programmation récursive¹ de l'UPMC. Cette carte de référence est librement utilisée lors des contrôles de connaissance.

Citation

<nom-fonction>: *<type-fonction>*
<nom-fonction> *<variable>**) sémantique...
 [ERREUR lorsque ...
 HYPOTHÈSE: ...

ne (*<nom-fonction>* *<variable>**)
)

Table des types

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

<type> → *<type>*
<type> * *<types>*

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

<destination> → *<type>*
 Rien

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

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

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

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

<le-type> → alpha
 beta
 etc.

Table du langage

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

<ant> → *<expression>*

<ative> → (if *<condition>* *<conséquence>* *<alternant>*)

<ation> → (*<fonction>* *<argument>**)

<ent> → *<expression>*

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

<en> → #t
 #f

<é> → " caractères autres que guillemets "

<n> → (quote *<S-expression>*)
 ' *<S-expression>*

<> → (*<condition>* *<expression>*)

<s> → *<clause>* *<clause>**
*<clause>** (else *<expression>*)

<ion> → *<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 précède 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** / ≠ 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** / ≠ 0 / → **int**

(quotient *n*₁ *n*₂) rend le quotient de la division euclidienne de *n*₁ par *n*₂

modulo: **int** × **int** / ≠ 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*))
 (caddr *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

length: LISTE[α] → nat
 (length *l*) rend la longueur de la liste *l*

liste: $\text{LISTE}[\alpha] \times \dots \rightarrow \text{LISTE}[\alpha]$
(liste $\ell \dots$) rend la concaténation des listes reçues en arguments
inverse: $\text{LISTE}[\alpha] \rightarrow \text{LISTE}[\alpha]$
(inverse ℓ) rend la liste renversée de ℓ
préfixe: $\alpha \times \text{LISTE}[\alpha] \rightarrow \text{LISTE}[\alpha] + \#f$
(préfixe $v \ell$) rend le suffixe de ℓ débutant par v ou $\#f$ si v n'apparaît pas dans ℓ
clé: $\alpha \times \text{LISTE}[\text{COUPLE}[\alpha \beta]] \rightarrow \text{COUPLE}[\alpha \beta] + \#f$
(clé $c \text{ al}$) rend le couple (clé valeur) dont la clé est c ou bien $\#f$ à pas de tel couple
test: $\text{Valeur} \rightarrow \text{bool}$
(test v) reconnaît que v est un symbole
chaîne: $\text{Valeur} \rightarrow \text{bool}$
(chaîne v) reconnaît que v est une chaîne de caractères
longueur: $\text{string} \rightarrow \text{nat}$
(longueur s) rend la longueur de la chaîne s
concat: $\text{string} \times \text{nat} \times \text{nat} \rightarrow \text{string}$
(concat $s i j$) construit la chaîne à partir des caractères $s i j$
concat: $\text{string} \times \dots \rightarrow \text{string}$
(concat $s \dots$) construit la concaténation de toutes les chaînes reçues en arguments
accès: $\text{Valeur} \rightarrow \text{bool}$
(accès v) reconnaît que v est un vecteur
accès: $\alpha \times \dots \rightarrow \text{VECTEUR}[\alpha]$
(accès $v \dots$) rend le vecteur dont les termes sont les arguments
accès: $\text{VECTEUR}[\alpha] \rightarrow \alpha$
(accès vecteur $index$) rend le $index$ -ième terme du vecteur
accès: $\text{VECTEUR}[\alpha] \rightarrow \text{nat}$
(accès vecteur) rend la longueur du vecteur
accès: $\text{VECTEUR}[\alpha] \rightarrow \text{LISTE}[\alpha]$
(accès vecteur) rend la liste des termes du vecteur
accès: $\text{LISTE}[\alpha] \rightarrow \text{VECTEUR}[\alpha]$
(accès liste) rend le vecteur des termes de la liste
accès: $(\alpha \rightarrow \beta) \times \text{LISTE}[\alpha] \rightarrow \text{LISTE}[\beta]$
(accès $f \ell$) rend la liste dont les termes sont les images par f des termes de la liste ℓ
affichage: $\text{Valeur} \rightarrow \text{Rien}$
(affichage v) imprime la valeur, ne rend rien d'intéressant
affichage: Rien
(affichage) passe à la ligne, ne rend rien d'intéressant

Opérations

opérations et formes spéciales suivantes ne sont utilisables sous DrScheme que si le paxon (ou *teachpack*) MIAStools est actif.
opérations: $(\alpha \rightarrow \text{bool}) \times \text{LISTE}[\alpha] \rightarrow \text{LISTE}[\alpha]$
(opérations $p \ell$) rend la liste extraite de ℓ dont les termes satisfont le prédicat p
opérations: $(\alpha \times \beta \rightarrow \beta) \times \beta \times \text{LISTE}[\alpha] \rightarrow \beta$
(opérations $f e \ell$) Compose binairement par f les éléments de la liste ℓ en commençant par e . Plus formellement: $(\text{reduce } f e (\text{list } e_1 \dots e_n)) \equiv (f e_1 (f e_2 \dots (f e_n e) \dots))$.
opérations: $\text{symbol} \times \alpha \times \dots \rightarrow \text{symbol}$
(opérations $f \dots$) 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?: $(\alpha \times \dots \rightarrow \beta) \times \alpha \times \dots \rightarrow \text{bool}$
(erreur? $f \dots$) teste si une fonction f appliquée à ses arguments provoque une erreur.

vérifier: $\dots \rightarrow \text{bool}$
(vérifier $\text{nom} \text{sexp} == \text{sexp} \dots$) 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: $\rightarrow \text{Image}$
(image-vide) rend une image carrée blanche.

filled-triangle: $\text{Coordonnée}^6 \rightarrow \text{Image}$
(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: $\text{Image} \rightarrow \text{Image}$
(invert $image$) rend une nouvelle image à couleur inversée.

line: $\text{Coordonnée}^4 \rightarrow \text{Image}$
(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: $\text{Image} \times \dots \rightarrow \text{Image}$
(overlay $image \dots$) rend une image superposant les contenus des images reçues en arguments [ERREUR lorsque les images sont de tailles différentes]

quarter-turn-right: $\text{Image} \rightarrow \text{Image}$
(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: $\text{Image} \times \text{Image} \rightarrow \text{Image}$
(stack $image_1 image_2$) rend une nouvelle image par juxtaposition de l'image₁ au dessus de l'image₂. [ERREUR lorsque les images sont de largeurs différentes]

image?: $\text{Valeur} \rightarrow \text{bool}$
(image? v) reconnaît si v est une image

image-width: $\text{Image} \rightarrow \text{nat}$
(image-width $image$) rend la largeur d'une image (en pixels).

image-height: $\text{Image} \rightarrow \text{nat}$
(image-height $image$) rend la hauteur d'une image (en pixels).

resize-image: $\text{Image} \times \text{nat} \times \text{nat} \rightarrow \text{Image}$
(resize-image $image \text{largeur} \text{hauteur}$) rend une nouvelle image homothétique avec une largeur et une hauteur spécifiées (en pixels).

mirror-image: $\text{Image} \rightarrow \text{Image}$
(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: $\text{LISTE}[\text{Ligne}] \rightarrow \text{Paragraphe}$
(paragraphe $lignes$) rend le paragraphe formé des lignes $lignes$
paragraphe-cons: $\text{Ligne} \times \text{Paragraphe} \rightarrow \text{Paragraphe}$
(paragraphe-cons $ligne \text{paragraphe}$) construit un paragraphe ayant en premier ligne suivi des lignes de paragraphe

lignes: $\text{Paragraphe} \rightarrow \text{LISTE}[\text{Ligne}]$
(lignes paragraphe) rend la liste des lignes que comporte le paragraphe

paragraphe-append: $\text{Paragraphe} \times \dots \rightarrow \text{Paragraphe}$
(paragraphe-append $\text{paragraphe} \dots$) rend le paragraphe correspondant à la concaténation des paragraphes reçus en arguments

->string: $\text{Valeur} \rightarrow \text{string}$
(->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: $\alpha \times \text{ArbreBinaire}[\alpha] \times \text{ArbreBinaire}[\alpha] \rightarrow \text{ArbreBinaire}[\alpha]$

(ab-noeud $etiquette \text{arbre} - \text{gauche} \text{arbre} - \text{droit}$) construit un arbre binaire

ab-vide: $\rightarrow \text{ArbreBinaire}[\alpha]$
(ab-vide) construit un arbre binaire vide

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

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

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

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

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

ag-noeud: $\alpha \times \text{LISTE}[\text{ArbreGeneral}[\alpha]] \rightarrow \text{ArbreGeneral}[\alpha]$
(ag-noeud $etiquette \text{sous} - \text{arbres}$) construit un arbre général

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

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

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