

—Grammaire des types—

Les éléments simples du langage (noms, symboles) sont en caractères machine, les ensembles d'éléments simples en caractères sans sérif. Les éléments composés sont entre < et >. L'étoile * signifie la répétition d'un élément et l'étoile ^ signifie le produit cartésien.

```
<type> ::= <type-base>
         ou <type-var>
         ou LISTE[<type>]
         ou ArbreBinaire[<type>]
         ou ArbreGeneral[<type>]
         ou COUPLE[<type> <type>]
         ou NUPLET[<type>*]
         ou (<type-fonc>)

<type-base> ::= nat ou int ou Nombre
              ou bool ou string
              ou Valeur ou Image ou void
              ou symbol

<type-var> ::= alpha ou beta etc.

<type-fonc> ::= -> <type>
              ou <type-args> -> <type>
              ou <type-args> -> <type> + #f

<type-args> ::= <type>
              ou <type> * <type-args>
              ou <type> * ...
              ou <type>^n

avec          n = 1, 2, 3, etc.
```

Le type **Nombre** comprend les entiers naturels (**nat**), les entiers relatifs (**int**) et les nombres flottants (avec point décimal).

Le type **void** correspond à des valeurs sans intérêt dont on ne peut rien faire.

Le type **Valeur** correspond à l'ensemble des valeurs possibles (nombre, booléen, chaîne de caractères, image, etc.).

—Grammaire du langage—

```
<prog> ::= <définition>
         ou <expression>
         ou <prog> <prog>

<définition> ::= (define ( nom-fonc <variables> )
                 <corps> )

<corps> ::= <définition>* <expression>

<variables> ::= variable*

<expression> ::= variable ou constante
               ou <application>
               ou <forme-spéciale>

<application> ::= (<expression> <expression>*)
```

```
<forme-spéciale> ::=
  (if <expression>
   <expression>
   <expression> )
  ou (and <expression>* )
  ou (or <expression>* )
  ou (cond <clauses> )
  ou (let ( <liaison>* )
         <corps> )
  ou (let* ( <liaison>* )
        <corps> )
  ou '<expression>
  ou (quote<expression>)
  ou (verifier nom-fonction <expression-test>*)

<clauses> ::= <clause>*
            ou <clause>* (else <expression> )

<clause> ::= ( <expression> <expression> )

<liaison> ::= ( variable <expression> )

<expression-test> ::= <expression> -> constante
                    <expression> => constante
                    ou <expression> -> ERREUR
```

Cette dernière expression de test permet de vérifier qu'une erreur est attendue mais on ne peut, par contre, spécifier la nature de l'erreur. Attention, si l'on ne peut écrire qu'une constante à droite de la flèche, on peut, en revanche, écrire n'importe quelle expression à gauche. **verifier** doit comporter au moins deux tests.

```
(cond (<expr1> <expr2>) <clause>*)
est équivalent à
(if <expr1> <expr2> (cond <clause>* ))
et, (cond (else <expression>))
est équivalent à <expression>.
```

```
(let* (<liaison> <liaison>* ) <corps>)
est équivalent à
(let (<liaison>) (let* (<liaison>* ) <corps>))
```

—Commentaires—

Ligne commençant par un point-virgule (;) au moins. Un commentaire sert à la documentation du code : il ne fait pas partie du code évalué (voir ci-dessous : spécification, etc.)

—Vocabulaire—

Caractères, symboles et mots clés réservés :
 ; () ' + - * / = < > " #f #t
and or if cond else let let* define verifier
 constante : les booléens **#f #t**, les nombres, les chaînes entre guillemets, les citations

variable, nom-fonc, symbole : tout ce qui n'est pas constante ni réservé

—Spécification, signature, définition—

```
;;; nom-fonc: <type-fonc>
;;; (nom-fonc <variables>) texte explicatif
;;; HYPOTHESE texte
;;; ERREUR texte
(define (nom-fonc <variables>)
  <définition>*
  <expression> )
```

—Fonction booléenne—

```
not: bool -> bool
(not b) rend la négation de b
```

—Prédicats sur les valeurs—

```
equal?: Valeur * Valeur -> bool
(equal? v1 v2) vérifie l'égalité de deux valeurs

number?: Valeur -> bool
(number? v) reconnaît si v est un nombre réel

integer?: Valeur -> bool
(integer? v) reconnaît si v est un nombre entier

boolean?: Valeur -> bool
(boolean? v) reconnaît si v est un booléen

string?: Valeur -> bool
(string? v) reconnaît si v est une chaîne de caractères

symbol?: Valeur -> bool
(symbol? v) reconnaît si v est un symbole

list?: Valeur -> bool
(list? v) reconnaît si v est une liste (possiblement vide)
```

—Prédicats sur les nombres—

```
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 y) vérifie que x et y sont égaux

<: Nombre * Nombre -> bool
(< x y) vérifie que x est strictement inférieur à y

<=: Nombre * Nombre -> bool
(<= x y) vérifie que x est inférieur ou égal à y
```

>: Nombre * Nombre -> bool
 (> x y) vérifie que x est strictement supérieur à y

>=: Nombre * Nombre -> bool
 (>= x y) vérifie que x est supérieur ou égal à y

—Fonctions arithmétiques—

+: Nombre * Nombre * ... -> Nombre
 (+ x1 x2 ...) rend la somme des arguments

-: Nombre -> Nombre
 (- x) rend l'opposé de x

-: Nombre * Nombre -> Nombre
 (- x y) rend la différence x - y

*: Nombre * Nombre * ... -> Nombre
 (* x1 x2 ...) rend le produit des arguments

/: Nombre * Nombre -> Nombre
 (/ x y) rend la division x / y
 ERREUR lorsque y est égal à 0

min: Nombre * Nombre * ... -> Nombre
 (min x1 x2 ...) rend le plus petit des arguments

max: Nombre * Nombre * ... -> Nombre
 (max x1 x2 ...) rend le plus grand des arguments

sqrt: Nombre -> Nombre
 (sqrt x) rend une valeur approchée de la racine carrée de x positif

quotient: int * int -> int
 (quotient a b) rend la division euclidienne de a par b
 ERREUR lorsque b est égal à 0

remainder: int * int -> int
 (remainder a b) rend le reste (du même signe que a) de la division euclidienne de a par b
 ERREUR lorsque b est égal à 0

modulo: int * int -> int
 (modulo a b) rend le reste (du même signe que b) de la division euclidienne de a par b
 ERREUR lorsque b est égal à 0

random: nat -> nat
 (random k) rend un nombre entier pseudo-aléatoire entre 0 et k-1

—Chaînes de caractères—

string-length: string -> nat
 (string-length s) rend la longueur (nombre de caractères) de la chaîne s

string-append: string * ... -> string

(string-append s1 ...) rend la chaîne obtenue en concaténant ses arguments

substring: string * nat * nat -> string
 (substring s i1 i2) rend la sous-chaîne de s commençant à l'indice i1 inclus et terminant à l'indice i2 exclus. Le premier indice est 0.

ERREUR lorsque i1 ou i2 n'est pas un indice dans s ou si i2 < i1

string>?: string * string -> bool
 (string>? s1 s2) vérifie si la chaîne s1 est strictement après s2 dans l'ordre du dictionnaire.

—Listes—

cons: alpha * LISTE[alpha] -> LISTE[alpha]
 (cons x L) rend la liste commençant par x et se poursuivant par les éléments de L

car: LISTE[alpha] -> alpha
 (car L) rend le premier élément de la liste L
 ERREUR lorsque L est vide

cdr: LISTE[alpha] -> LISTE[alpha]
 (cdr L) rend la liste L privée de son premier élément
 ERREUR lorsque L est vide

list: alpha * ... -> LISTE[alpha]
 (list x1 ...) rend la liste de ses arguments.
 (list) sans argument, rend la liste vide

pair?: Valeur -> bool
 (pair? x) rend #t si x est une liste non vide; qui a un car et un cdr

append: LISTE[alpha] * ... -> LISTE[alpha]
 (append L1 ...) rend la liste composée des éléments de ses arguments (concaténation)

assoc: alpha * LISTE[COUPLE[alpha beta]]
 -> COUPLE[alpha beta] + #f
 (assoc x L) rend la première association de L dont x est la clé ou #f si aucune association n'a la clé x

map: (alpha -> beta) * LISTE[alpha]
 -> LISTE[beta]
 (map f L) rend la liste ((f x1) ... (f xn)) si L est la liste (x1 ... xn)

filter: (alpha -> bool) * LISTE[alpha]
 -> LISTE[alpha]
 (filter p L) rend la liste des éléments de L qui vérifient le prédicat p

reduce:
 (alpha * beta -> beta) * beta * LISTE[alpha]
 -> beta

(reduce f e L) rend la valeur de (f x1 (f x2 ... (f xn e)...)) si L est la liste (x1 x2 ... xn)

—Arbres binaires—

ab-vide: -> ArbreBinaire[alpha]
 (ab-vide) sans argument, rend l'arbre binaire vide

ab-noeud:
 alpha * ArbreBinaire[alpha] * ArbreBinaire[alpha]
 -> ArbreBinaire[alpha]
 (ab-noeud x A1 A2) rend l'arbre de racine d'étiquette x, de sous-arbre gauche A1 et de sous-arbre droit A2

ab-etiquette: ArbreBinaire[alpha] -> alpha
 (ab-etiquette A) rend l'étiquette de la racine de A
 ERREUR lorsque A est vide

ab-gauche:
 ArbreBinaire[alpha] -> ArbreBinaire[alpha]
 (ab-gauche A) rend le sous-arbre gauche de A
 ERREUR lorsque A est vide

ab-droit:
 ArbreBinaire[alpha] -> ArbreBinaire[alpha]
 (ab-droit A) rend le sous-arbre droit de A
 ERREUR lorsque A est vide

ab-noeud?: ArbreBinaire[alpha] -> bool
 (ab-noeud? A) rend #t si A n'est pas vide

ab-affiche: ArbreBinaire[alpha] -> void
 (ab-affiche A) affiche l'arbre A

—Arbres généraux et forêts—

Foret[alpha] = LISTE[ArbreGeneral[alpha]]

ag-noeud: alpha * Foret[alpha]
 -> ArbreGeneral[alpha]
 (ag-noeud x F) construit l'arbre général de racine d'étiquette x et de forêt F

ag-etiquette: ArbreGeneral[alpha] -> alpha
 (ag-etiquette A) rend l'étiquette de la racine de A

ag-foret: ArbreGeneral[alpha] -> Foret[alpha]
 (ag-foret A) rend la forêt des sous-arbres de A

ag-affiche: ArbreGeneral[alpha] -> void
 (ag-affiche A) affiche l'arbre A

—Erreur—

erreur: Valeur * ... -> void
 (erreur v1 ...) interrompt le programme et affiche les valeurs passées en arguments