Support du cours LI101
Programmation récursive

Titou Durand

Ce document est une version rédigée du MOOC « Programmation récursive ». Il est dû à Titou Durand avec l'aide des autres enseignants de ce cours : Anne Brygoo, Pascal Manoury, Maryse Pelletier, Christian Queinnec et Michèle Soria.

Ce document a été écrit, en son temps, pour le cours LI101 du DEUG MIAS de l'UPMC. Il a servi de base pour un livre [Brygoo et al., 2004] mais il est toujours approprié pour le MOOC « Programmation récursive ».

Bonne lecture !

Références

[Brygoo et al., 2004] Brygoo, A., Durand, T., Pelletier, M., Queinnec, C., and Soria, M. (2004). *Programmation récursive (en Scheme)*. Dunod.

Première saison

Version 1.12

Sommaire

1. Introduction

1.1. Qu'est-ce-que l'informatique ?

D'après LE PETIT ROBERT (les mots sont soulignés par nous) :

«Informatique : (1962) Science du traitement de l'information ; ensemble des techniques de la collecte, du tri, de la mise en mémoire, du stockage, de la transmission et de l'utilisation des informations traitées automatiquement à l'aide de programmes (=> logiciel) mis en œuvre sur ordinateurs.»

Faisons quelques remarques à propos de cette définition :

1.1.1. Science — technique

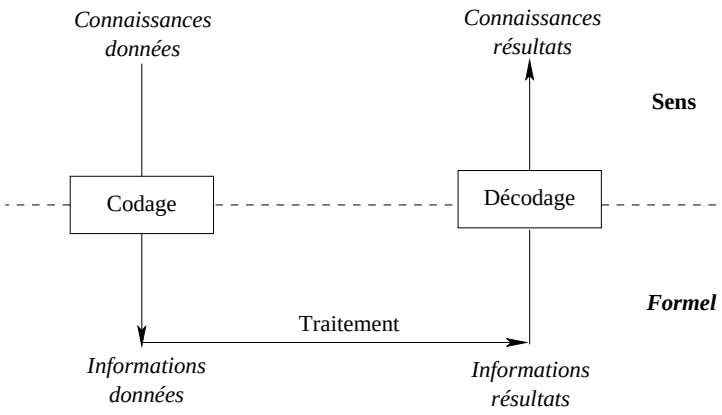
Comme pour le mot optique, le mot informatique a deux sens : c'est une science et c'est une technique. Mais alors, doit-on enseigner la science ou la technologie (d'après LE PETIT ROBERT, le sens de ce mot est « Théorie générale et études spécifiques (outils, machines, procédés...) des techniques. ») ? En Deug Mias, nous enseignerons les deux aspects, le premier expliquant le second et le second rendant plus concret le premier.

1.1.2. Information

On peut partir du sens commun (écouter les informations...) et faire quelques remarques :
- on consulte un bulletin d'informations pour apprendre des connaissances nouvelles ;
- si c'est un bulletin en japonais...
- certains livres d'informatique ne nous apprennent rien, car nous savons ce qu'ils disent.

Ainsi, il ne faut pas confondre connaissance et information, même si ces deux notions sont liées. En fait, il faut séparer la forme (qui peut être du texte, des images, des sons...) de sa signification et le mot information correspond à la forme : en informatique, nous manipulons du formel.

Cette information peut être traitée (manipulée). Par exemple, si on connait le nombre d'étudiants inscrits en DEUG MIAS première année et le nombre maximal d'étudiants par groupe de TD, on peut coder ces connaissances par deux nombres écrits en base dix et on peut, par un traitement formel, calculer combien il faut de groupes de TD. On peut représenter ce processus par le schéma suivant :



Dans le sens commun, nous avons vu que nous parlions d'information nulle (pour dire que l'information n'apporte rien). Ainsi, il semblerait que l'on puisse quantifier l'information. C'est bien le cas comme l'a montré Shannon dans sa théorie de l'information.

1.1.3. Traitement sur ordinateur de l'information (logiciels)

« informations traitées automatiquement à l'aide de programmes (=> logiciel) mis en œuvre sur ordinateurs »

Système d'exploitation : il existe, parmi les programmes présents dans un ordinateur, des programmes qui permettent de gérer ce dernier. Par exemple, lorsque vous appuyer sur une touche du clavier, vous envoyez une information à l'ordinateur et il faut que ce dernier traite cette information. Il est clair que ces programmes sont fondamentaux puisque tout passe par eux. L'ensemble de ces programmes constitue ce que l'on appelle le **système d'exploitation**.

En DEUG, nous travaillerons sous deux systèmes d'exploitation :

– Unix (Linux) au premier semestre,
– Windows au second semestre.

Logiciels : pour manipuler des informations à l'aide d'un ordinateur, il faut une description précise du traitement à effectuer : c'est un **algorithme**. De plus, tout algorithme doit être écrit dans un langage « compréhensible par les ordinateurs » : c'est un **programme**.

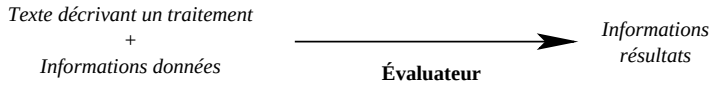
1.2. Processus d'évaluation

En fait, il n'existe pas de langage compréhensible par les ordinateurs !

– il existe des langages machines qui sont compréhensibles par **un** type d'ordinateur,
– … et complètement incompréhensibles par l'humain.

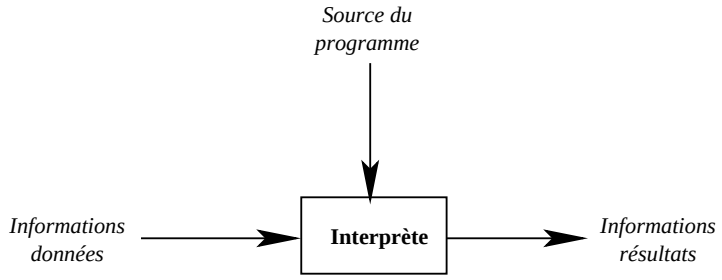
Définition : le **processus d'évaluation** permet de produire de l'information (informations résultats) à partir

– d'un texte qui décrit un traitement (on dit le **source du programme**),
– d'informations consommées (informations données).

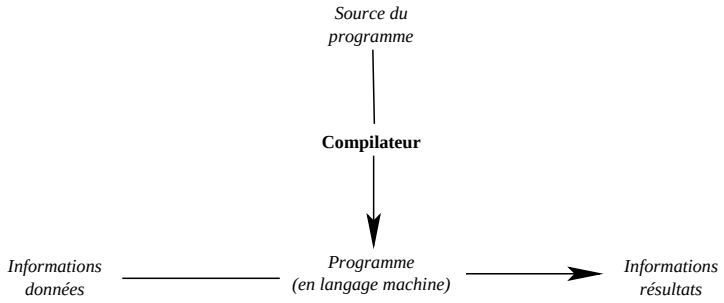


Le processus d'évaluation se décline sous deux formes :

1.2.1. Interprète



1.2.2. Compilateur



1.2.3. En plus...

Remarquons qu'avec la défi nition que nous avons donnée, ce processus d'évaluation est aussi à la base de la compréhension de ce que sont les évaluateurs (interprètes ou compilateurs) inclus dans les applicatifs (texteur, tableur...).

1.3. Scheme

Pour concrétiser les notions que nous voulons étudier dans ce cours, nous utiliserons un langage de programmation (Scheme) et, pour «passer sur machine», nous utiliserons un environnement Scheme, qui comporte, entre autres, un évaluateur (DrScheme).

Un programme Scheme – rappelons que c'est un texte – est appelé *expression*. Autrement dit, une *expression* est un programme qui, après exécution, fournira une valeur (unique).

1.3.1. Les commentaires

Avant d'expliquer comment sont écrites les expressions, nous voudrions parler des ***commentaires*** : les sources des programmes doivent être analysés par l'ordinateur (cf. processus d'évaluation) mais ils doivent aussi être lus par des humains (l'enseignant en ce qui vous concerne, d'autres membres de l'équipe dans l'industrie et, dans tous les cas, le programmeur lui-même). Pour que ces sources soient lisibles (compréhensibles) par des humains, il est indispensable de rajouter du texte qui aide l'humain et qui n'est pas pris en compte par l'ordinateur : ce sont des commentaires.

En Scheme, un point-virgule indique le début d'un commentaire qui continue jusqu'à la fin de la ligne. Par exemple :

```
;;; aire-disque: Nombre -> Nombre
;;; (aire-disque r) rend l'aire d'un disque
;;; de rayon r
(define (aire-disque r)
  (* 3.1416 r r)) ; ou encore (* 3.1416 (* r r)))
```

Notons enfin que, par convention, le nombre de point-virgules indique le genre de commentaire. Nous préciserons ces conventions dans les cours suivants.

1.3.2. Les objets primitifs

Le langage permet de manipuler les constantes entières, flottantes, chaînes de caractères, booléennes... – par exemple la constante entière `42`, flottante `2.3`, la constante chaîne de caractères `"Anne"`, la valeur booléenne vraie `#t`, la valeur booléenne faux `#f` (qui sont affichées, dans de nombreux niveaux de DrScheme, comme `true` et `false`) et, aussi des fonctions primitives comme `+`, `*` ...

Pour s'auto-évaluer
Exercices d'assouplissement[1]
Questions de cours[2]

2. Étude des expressions

En mathématique ou en informatique, une expression exprime la façon de calculer une valeur. Par exemple,

$a * f(a-1)$

dit qu'il faut appliquer la multiplication – qui est représentée par l'opérateur $*$ – aux valeurs des deux sous-expressions a et $f(a-1)$. À nouveau, pour cette dernière, on doit appliquer la fonction f à la valeur de $a-1$ qui est elle-même obtenue...

Première remarque : La dénomination « programmation applicative » qui désigne le style de programmation que nous utiliserons pendant ce semestre est issue de cette vision du calcul d'une expression (« appliquer la multiplication », « appliquer la fonction f »...). En effet, l'évaluation des expressions est une des clefs d'une telle programmation.

Remarque historique : Fortran (FORmula TRANslator), premier langage de programmation évolué et compilé, a été défini, en 1954, par John Backus dont un des buts était de « permettre une formulation concise d'un problème en utilisant les notations mathématiques » : il s'agissait avant tout de pouvoir écrire directement les **expressions** mathématiques.

[1] `http://127.0.0.1:20022/q-ab-introduction-1.quizz`
[2] `http://127.0.0.1:20022/q-ab-introduction-2.quizz`

2.1. Compréhension d'une expression

Considérons l'expression, écrite dans le langage mathématique habituel,

$3x^2 + 2x + 4$

On peut déjà remarquer quelques caractéristiques qui ne sont jamais présentes dans les langages informatiques :

- dans $3x^2$ et dans $2x$, la multiplication est sous-entendue, c'est-à-dire que l'on peut écrire aussi $3 \times x^2 + 2 \times x + 4$;
- l'exponentiation (dans x^2) est notée en mettant l'exposant en dessus de la ligne, ce qui n'est pas possible dans les écritures linéaires utilisées en informatique.

Il faut remarquer que l'addition, la multiplication, la division... sont des opérations binaires : à deux valeurs elles font correspondre une troisième. Lors de l'évaluation d'une expression, une telle opération (qui est appelée l'opération principale de l'expression) est appliquée aux résultats de l'évaluation de deux sous-expressions. Pour bien indiquer les sous-expressions, on peut entourer chaque expression, non réduite à une variable, par des parenthèses : on dit que l'on a un langage complètement parenthésé. Mais habituellement on « oublie » des parenthèses :

$a-b+c$	peut se comprendre	$(a-b)+c$
	ou	$a-(b+c)$
$a+b\times c$	peut se comprendre	$(a+b)\times c$
	ou	$a+(b\times c)$
$a/b/c$	peut se comprendre	$(a/b)/c$
	ou	$a/(b/c)$

ce qui, dans les trois cas, ne donne pas le même résultat ! Quelle est la bonne lecture ? => difficultés pour analyser automatiquement les expressions.

2.2. Différentes écritures linéaires d'une expression

Il existe plusieurs écritures linéaires (*i.e.* comme suite de caractères) d'une expression selon le placement de l'opérateur par rapport à ses opérandes :

- en préfixé, l'opérateur précède ses opérandes,
- en suffixé, l'opérateur suit ses opérandes,
- en infixé, un opérateur binaire se situe entre ses arguments.

Par exemple, l'expression

$a \ / \ c \ + \ 5sin(b \times d)$, peut s'écrire :

en préfixé : $+/ac \times 5sin \times bd$

en suffixé : $ac/5bd \times sin \times +$

en infixé : $((a \ / \ c) \ + \ (5 \times sin(b \times d)))$

Lemme de Lukasiewicz : si l'on connait l'arité des opérateurs, on peut retrouver l'expression à partir de l'une des notations suffixe et préfixe.

Autrement dit, ces deux notations représentent de façon non-ambiguë l'expression. En revanche, la notation infixe est ambiguë et il est nécessaire d'utiliser des parenthèses.

Remarques :

1. la notation mathématique utilise la notation préfixe pour les fonctions en écrivant d'abord le nom de la fonction puis, entre parenthèses, les différents arguments séparés par une virgule ;
2. noter aussi la différence entre *ab* qu'on lit $a \times b$ – on considère donc qu'il y a deux symboles, *a* et *b* – et *sin* qui représente un seul symbole, le nom de la fonction sinus ;
3. dans la terminologie mathématique, on distingue les opérateurs ($+$, $\times$...) et les fonctions (*sin*...) ; en Scheme, on ne parlera pas d'opérateur mais systématiquement de fonctions ;
4. en Scheme, nous nommerons ***application*s** ces expressions (car nous verrons que la notion d'expression est plus générale : une application Scheme est une expression Scheme, mais il y a des expressions Scheme qui ne sont pas des applications) ;
5. dans la littérature, on dit parfois « appel de fonction » à la place d'« application de fonction ».

3. Écriture Scheme d'une application

Caractéristiques des applications *en Scheme* :

- écriture préfixe,
- complètement parenthésée,
- les parenthèses entourent toute l'expression (*i.e.* la fonction et les arguments),
- le séparateur est l'espace.

Exemples :

```
(+ 3 a)
```

On peut imbriquer les applications :

```
(+ (/ a c) (* 5 (sin  (* b d))))
```

Remarque : les fonctions peuvent alors avoir un nombre quelconque d'arguments. Par exemple :

```
(+ (* 3 x x) (* 5 x) 8)
```

On peut écrire les expressions sur plusieurs lignes, ce qui améliore la lisibilité :

```
(+ (* 3 x x)
   (* 5 x)
   8)
```

Attention, ne pas confondre l'entier relatif -3 avec l'expression `(- 0 3)` ou l'expression `(- 3)`, qui ont la même valeur, et avec l'expression `(- 3 0)`, qui vaut 3 et `(-3)` (sans espace entre le tiret et le 3) qui donne une erreur, `-3` n'étant pas une fonction.

`(- 0 3)` → -3
`(- 3)` → -3
`(- 3 0)` → 3
`(-3)` → ERREUR

3.1. Terminologie

Dans une application, le premier élément est la ***fonction*** (et, comme nous le verrons plus tard, c'est elle-même une expression) que l'on applique à des ***argument*s** (qui sont eux-mêmes des expressions). Par exemple, dans l'expression

```
(+ (* 3 7 7) (* 5 7) 8)
```

- la fonction est `+` ;
- les arguments sont `(* 3 7 7)`, `(* 5 7)` et `8`.

3.2. Grammaire

On définit comment doit être écrite une application en utilisant une *règle de grammaire* :

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

Dans une telle règle de grammaire :

- *application*, qui est avant la flèche et est écrit entre chevrons (< et >), est l'*unité syntaxique* que l'on est en train de définir ;
- *fonction* et *argument*, qui sont écrits entre chevrons (< et >), sont des unités syntaxiques définies par une (autre ou celle que l'on est en train d'écrire) règle de grammaire ;

[3] `http://127.0.0.1:20022/q-ab-expression-1.` `quizz`
[4] `http://127.0.0.1:20022/q-ab-expression-2.` `quizz`

(et *j* qui ne sont pas écrits entre chevrons sont des *terminaux* et seront écrits tels quels dans les sources Scheme.

Cette règle de grammaire dit qu'une application est constituée par une parenthèse ouvrante suivie d'une fonction puis d'un nombre quelconque, éventuellement 0, (c'est le sens de l'étoile) d'arguments et enfi n d'une parenthèse fermante.

En fait, les arguments sont des expressions quelconques (c'est ce qui permet, entre autres, de pouvoir imbriquer les applications) et il en est de même pour les fonctions :

<argument> → *<expression>*
<fonction> → *<expression>*

3.3. Notions de syntaxe et de sémantique

Dans le paragraphe précédent, nous avons défi ni comment doit être écrite une application pour qu'elle puisse être « comprise » par l'interprète Scheme. On dit que l'on a décrit la syntaxe du langage.

Il faut également savoir ce que représente une telle application, autrement dit quelle est sa valeur. On dit que l'on défi nit la sémantique du langage. En effet, la valeur de l'application calculée par l'interprète doit correspondre au *sens* que l'humain, qui lit ou écrit le source du programme, donne à cette application.

Dans ce cours, nous donnerons trois formes de sémantiques :

- tout d'abord une sémantique naïve où l'on décrit, « avec les mains », la valeur que doit affi cher un interprète Scheme,
- cette vision ne permettant pas de comprendre ce qui se passe lorsque l'expression est complexe, nous donnerons ensuite une sémantique plus formelle qui nous permettra de défi nir précisément la valeur des expressions que nous écrirons en DEUG,
- enfi n, l'interprète que nous écrirons dans la troisième partie de ce cours est une autre forme de sémantique (on indique bien ainsi ce que doit valoir une expression), bien sûr de nature très différente.

Remarque : la sémantique formelle que nous donnerons par la suite est suffi sante pour décrire le sens des expressions Scheme que nous écrirons en DEUG. Elle n'est pas assez puissante pour décrire le sens de n'importe quelle expression Scheme.

3.4. Évaluation d'une application

Pour évaluer une application, on évalue chacun de ses arguments (ce qui nous donne des valeurs) et on applique la fonction à ces valeurs. Par exemple, pour évaluer l'application

```
(+ (* 3 7 7) (* 5 7) 8)
```

1 - on évalue les arguments :

`(* 3 7 7)` → `147`

`(* 5 7)` → `35`

`8` → `8`

2 - on applique la fonction + aux arguments 147, 35 et 8 :

`(+ (* 3 7 7) (* 5 7) 8)` → `190`

On peut visualiser ce processus de calcul en utilisant DrScheme et en faisant évaluer l'expression en mode « pas à pas ».

Pour s'auto-évaluer
Exercices d'assouplissement[5]
Questions de cours[6]

[5] `http://127.0.0.1:20022/q-ab-application-1.quizz`
[6] `http://127.0.0.1:20022/q-ab-application-2.quizz`

4. Définition de fonctions

*définition de fonction

Dans les exemples précédents, nous avons utilisé des fonctions fournies par DrScheme (on dit alors qu'elles sont prédéfinies). Notons que ces fonctions sont de deux genres :

- des fonctions qui sont implantées dans l'interpréteur – c'est le cas des fonctions que nous avons utilisées jusqu'alors –, on parle alors de fonctions de base ou de **primitives** ;
- des fonctions qui ont été écrites en Scheme, par les écrivains de l'interpréteur ou par d'autres personnes, qui ont été regroupées par centre d'intérêt et qui sont mises à disposition de l'utilisateur, à condition que celui-ci le demande. On parle alors d'**unités de bibliothèque** et on dit que l'utilisateur demande le chargement d'une unité de bibliothèque. Par exemple, le DrScheme qui vous est fourni possède une entrée dans le menu MIAS qui peut charger l'unité de bibliothèque *mias*, écrite par nos soins et qui comporte des fonctions, non présentes dans Scheme, utiles pour ce cours.

On peut aussi définir ses propres fonctions à l'aide de **définitions**. C'est un mécanisme d'abstraction qui permet d'associer un « programme » à un nom.

4.1. Rappels mathématiques

Voici un énoncé « mathématique » :

Soit f la fonction qui associe à un nombre r le nombre πr^2.
Calculer $f(22)$ et $f(\sqrt{2})$.

La première phrase est une *définition* de la fonction f, définition que l'on peut noter plus formellement par :

$$\begin{array}{rcl} f : Nombre & \rightarrow & Nombre \\ r & \mapsto & \pi r^2 \end{array}$$

La seconde phrase demande d'évaluer deux *applications* de la fonction f : d'abord appliquer f avec l'*argument* 22, ensuite appliquer f avec l'argument $\sqrt{2}$ (qu'il faut d'abord évaluer).

On peut ensuite définir une autre fonction, par exemple g :

$$\begin{array}{rcl} g : Nombre \times Nombre & \rightarrow & Nombre \\ (r,h) & \mapsto & f(r) \times h \end{array}$$

et demander d'évaluer $g(22,5)$...

4.1.1. Notions de type et de signature

Reprenons la première ligne de la définition formelle de la fonction g :

$g : Nombre \times Nombre \rightarrow Nombre$

Cette partie de la définition est appelée la **signature** de la fonction :

$g : Nombre \times Nombre \rightarrow Nombre$

Signature de la fonction

La signature de la fonction est composée du nom de la fonction suivi d'un caractère deux-points et du **type** de la fonction :

$g : \underbrace{Nombre \times Nombre \rightarrow Nombre}_{\text{Type de la fonction}}$

Signature de la fonction

[7] http://127.0.0.1:20022/q-ab-application-3 .quizz

le type de la fonction étant constitué par les types de ses arguments (séparés par des $\times$), suivi du signe $\rightarrow$ et du type de son résultat.

4.1.2. Spécification d'une fonction (premier regard)

Supposons que l'on veuille calculer le poids au mètre d'un tube en fer, de rayon extérieur 49mm et d'épaisseur 1mm (le fer est de densité 7,87).

On peut utiliser la fonction *g* à condition que l'on ait remarqué qu'elle calculait le volume d'un cylindre, de rayon le premier argument de la fonction et de hauteur le second argument de la fonction. Notons bien que l'expression de calcul de la fonction *g* (que l'on utilise ou non la fonction f, que l'on écrive $\pi r^2 h$ ou $h\pi r^2$...) nous importe peu pour calculer le poids au mètre d'un tube en fer ; tout ce qui nous intéresse c'est que cette fonction calcule le volume d'un cylindre, de rayon le premier argument de la fonction et de hauteur le second argument de la fonction. Cette information est la **spécification** de la fonction et, si l'on veut réutiliser une fonction, on doit donner sa spécification :

g : *Nombre* $\times$ *Nombre* $\rightarrow$ *Nombre*
$g(r,h)$ renvoie le volume du cylindre de rayon *r* et de hauteur *h*.

voici l'expression pour calculer le poids au mètre :

$$7.87 \times (g(0.49,\ 10) - g(0.48,\ 10))$$

4.2. En Scheme

Dans le premier énoncé précédent, pour la fonction f, vous avez reconnu le calcul de l'aire d'un disque, aussi prenons-nous un identificateur plus « parlant ».

```
;;;  aire-disque : Nombre -> Nombre
;;;  (aire-disque r) rend la surface du disque de rayon «r»
(define   (aire-disque    r)
  (* 3.1416   (* r r)))  ; ou (* 3.1416 r r)
```

et on peut appliquer cette fonction :

```
;;;  premier essai de aire-disque (rend 1520.5344) :
   (aire-disque    22)  →  1520.5344
;;;  second essai de aire-disque (rend 6.283200000000002) :
   (aire-disque    (sqrt  2))  →  6.283200000000002
```

Autre exemple :

```
;;;  volume-cylindre : Nombre * Nombre -> Nombre
;;;  (volume-cylindre r h) rend le volume du cylindre de
;;;  rayon «r» et de hauteur «h»
(define   (volume-cylindre      r h)
  (* 3.1416   r r h))  ; ou (* (aire-disque r) h)
```

et on peut appliquer cette fonction :

```
;;;  essai de volume-cylindre (rend 7602.6720000000005) :
   (volume-cylindre      22  5)  →  7602.6720000000005
```

Un dernier exemple :

```
;;;   negative? : Nombre -> bool
;;;   (negative? x) rend « true » ssi «x» est strictement négatif
(define   (negative?    x)
  (< x 0))
```

fonction que l'on doit aussi essayer :

```
;;;  essai de negative? :
   (negative?    -5)  →  #T
   (negative?    0)  →  #F
   (negative?    5)  →  #F
```

Définition : un **prédicat** est une fonction qui rend un booléen.

Convention : les prédicats ont des noms qui se terminent par un point d'interrogation.

4.2.1. Grammaire des définitions Scheme

Une définition de fonction suit la règle de grammaire suivante :

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

Cette règle de grammaire comporte une étoile derrière l'unité syntaxique *<variable>*. Cela indique qu'il y a, dans les mots générés par cette règle de grammaire, un nombre quelconque – éventuellement 0 – d'éléments de cette unité syntaxique.

Ainsi une définition de fonction débute par une parenthèse ouvrante et le mot clef « *define* » puis entre parenthèses il y a le nom de la fonction suivi d'un nombre quelconque – éventuellement 0 – de *variable*s, différentes deux à deux, nécessaires à la fonction (autant de variables qu'il y aura d'arguments lors de l'application de la fonction) et enfin du corps de la fonction qui exprime comment on doit calculer ce qu'elle rend et qui répond à la règle de grammaire suivante :

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

Cette règle de grammaire est écrite sur deux lignes. Cela indique que l'on peut prendre l'une ou l'autre des possibilités : un corps est une définition suivie d'un corps ou une expression suivie d'un nombre quelconque (éventuellement 0) d'expressions (ou encore un nombre quelconque non nul d'expressions).
Remarque : dans ce cours, dans un premier temps, le corps sera une expression.

4.2.2. Écriture de la spécification

Systématiquement, pour toute définition de fonction, on écrit sa spécification sous forme de commentaires placés avant la définition, chacune des lignes de commentaire débutant par trois points-virgules :

- la première ligne (éventuellement les premières lignes pour des raisons de présentation) de la spécification comporte la signature de la fonction, le signe → étant remplacé par `->` et le signe × étant remplacé par `*` ;
- les lignes suivantes expliquent, en français, ce que renvoie la fonction ; pour faciliter les explications, cette partie débute par une application de la fonction aux variables utilisées dans la définition de cette dernière (par exemple, pour la fonction `aire-disque`, la définition commence par `(define (aire-disque r)` et l'explication commence par *(aire-disque r) rend* :

```
...
;;; aire-disque : Nombre -> Nombre
;;; (aire-disque r) rend la surface du disque de rayon «r»
(define (aire-disque r)
...
```

Il y a de nombreux exemples de spécifications ainsi écrites dans la carte de référence[8].

4.2.3. Évaluation d'une définition

Intuitivement – rappelons que nous aurons une vision plus formelle ultérieurement –, l'évaluation d'une définition ajoute une fonction

- que le programmeur peut utiliser dans l'écriture d'une application, sous réserve qu'il écrive le bon nombre d'arguments (à savoir le nombre des variables de la définition de la fonction),
- en mémorisant comment on doit évaluer une telle application (c'est le rôle du corps de la définition).

Pour s'auto-évaluer
Exercices d'assouplissement[9]
Questions de cours[10]
Approfondissement[11]

[8] http://www.licence.info.upmc.fr/lmd/licence/2005/ue/prec-2005oct/reference.ps
[9] http://127.0.0.1:20022/q-ab-definition-1.quizz
[10] http://127.0.0.1:20022/q-ab-definition-2.quizz
[11] http://127.0.0.1:20022/q-ab-definition-3.quizz

5. Alternative et conditionnelle

5.1. Alternative

*alternative

5.1.1. Exemples

Les expressions peuvent être définies par cas au moyen d'alternatives. Par exemple :

```
;;; valeur-absolue : Nombre -> Nombre
;;; (valeur-absolue x) rend la valeur absolue de «x»
(define (valeur-absolue x)
  (if (>= x 0)
      x
      (- x)))
```

Notons que nous pourrions aussi utiliser le prédicat `negative?` que nous avons définie ci-dessus :

```
;;; valeur-absolue-bis : Nombre -> Nombre
;;; (valeur-absolue-bis x) rend la valeur absolue de «x»
(define (valeur-absolue-bis x)
  (if (negative? x)
      (- x)
      x))
```

5.1.2. Grammaire

La syntaxe d'une alternative suit la grammaire suivante :

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

<condition> → *<expression>*

<conséquence> → *<expression>*

<alternant> → *<expression>*

Remarques :

- Une condition est une expression ayant pour valeur, soit vrai (noté `#t`), soit faux (noté `#f`).
- Une conséquence et un alternant sont des expressions quelconques.
- La règle de grammaire qui définit l'alternative est écrite sur deux lignes et on peut donc prendre l'une ou l'autre des possibilités. Dans un premier temps, nous n'utiliserons que la deuxième.

5.1.3. Évaluation d'une alternative

Une alternative s'évalue ainsi : la condition est d'abord évaluée. Si sa valeur est vraie, alors seulement la conséquence est évaluée (et la valeur de l'alternative est égale à la valeur de cette évaluation) sinon seulement l'alternant est évalué.

Ainsi, contrairement aux applications, pour une alternative, on ne commence pas par évaluer tous ses composants : on dit qu'une alternative est une ***forme spéciale***.

5.2. Écriture des conditions

Les conditions peuvent être construites avec les opérations logiques suivantes :

Tout d'abord la négation : il existe une fonction prédéfinie `not` :

```
;;; valeur-absolue-ter : Nombre -> Nombre
;;; (valeur-absolue-ter x) rend la valeur absolue de «x»
(define (valeur-absolue-ter x)
  (if (not (negative? x))
      x
      (- x)))
```

Remarque : la fonction `not`, qui rend un booléen, n'est pas un prédicat, mais une opération car elle a comme domaine un booléen (ainsi, elle a le même status que, par exemple, l'opération moins unaire sur les entiers – comme dans l'application `(- 3)`).

On peut écrire aussi des conjonctions et des disjonctions en suivant les règles de grammaire suivantes :

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

La conjonction et la disjonction ne sont pas des fonctions – comme la fonction `not` – mais des formes spéciales – comme l'alternative :

Pour évaluer `(and e1 e2 e3)`,

1. on évalue l'expression `e1` : si elle est fausse, toute l'expression est fausse et on n'évalue pas `e2` et `e3` ;
2. si `e1` est vraie, on évalue l'expression `e2` : si elle est fausse, toute l'expression est fausse et on n'évalue pas `e3` ;
3. si `e1` et `e2` sont vraies, on évalue l'expression `e3` : si elle est fausse, toute l'expression est fausse et si elle est vraie, toute l'expression est vraie.

De même, pour évaluer `(or e1 e2)`,

1. on évalue l'expression `e1` : si elle est vraie, toute l'expression est vraie et on n'évalue pas `e2` ;
2. si `e1` est fausse, on évalue l'expression `e2` : si elle est vraie, toute l'expression est vraie et si elle est fausse, toute l'expression est fausse.

5.3. Conditionnelle

*conditionnelle

L'alternative permet d'écrire des expressions dont la valeur est spécifi ée selon deux cas. S'il y a plus de deux cas, on écrit une alternative dont la conséquence ou l'alternant est une alternative.

5.3.1. Exemples

Considérons, par exemple, la fonction `signe` spécifi ée par :

```
;;; signe : Nombre -> nat
;;; (signe x) rend -1, 0 ou +1 selon que «x» est négatif, nul ou positif
```

On peut écrire une défi nition en utilisant des alternatives :

```
(define (signe x)
  (if (< x 0)
      -1
      (if (= x 0)
          0
          1)))
```

Mais, au lieu d'utiliser une cascade d'alternatives, il est préférable, pour la lisibilité, d'utiliser une conditionnelle :

```
(define (signe x)
  (cond ((< x 0) -1)
        ((= x 0) 0)
        (else 1)))
```

5.3.2. Grammaire

La syntaxe d'une conditionnelle suit la grammaire suivante :

<conditionnelle> → `(cond` *<clauses>* `)`
<clauses> → *<clause>* *<clause>**
<clause>* `(else` *<expression>* `)`
<clause> → `(` *<condition>* *<expression>* `)`

Remarques :

– La règle de grammaire qui définit <*clause*> est écrite sur deux lignes et on peut donc prendre l'une ou l'autre des possibilités. Nous n'utiliserons que la deuxième (et la dernière clause est donc un *else*).
– Noter bien les parenthèses : il y a un couple de parenthèses pour le `cond` , ce mot clef étant suivi de <*clauses*> qui est une suite de <*clause*>s, chacune des <*clause*>s étant constituée par un couple, entouré de parenthèses, <*condition*>, <*expression*>. Ainsi, une conditionnelle commence par `(cond (`<*condition*> et comme la condition est, la plupart du temps, une expression non réduite à une constante, elle commence elle-même par une parenthèse et il y a donc deux parenthèses ouvrantes après `cond` .

5.3.3. Évaluation d'une conditionnelle

Une conditionnelle s'évalue ainsi :
– la condition de la première clause est d'abord évaluée ; si sa valeur est vraie, alors seulement l'expression de cette clause est évaluée (et la valeur de la conditionnelle est égale à la valeur de cette évaluation) ;
– sinon, la condition de la deuxième clause est évaluée ; si sa valeur est vraie, l'expression de cette clause est évaluée (et la valeur de la conditionnelle est égale à la valeur de cette évaluation) ;
– ...
– si les évaluations de toutes les conditions rendent faux, la valeur de la conditionnelle est égale à l'évaluation de l'expression qui suit le `else` .

5.3.4. Conditionnelles et alternatives

Les conditionnelles ne sont pas essentielles, car elles peuvent être réécrites sous la forme d'alternatives :

```
(cond (else e)) ⇒ e
(cond (c1 e1) (else e2))) ⇒ (if c1 e1 e2)
(cond (c1 e1) (c2 e2) ... )) ⇒ (if c1 e1 (if c2 e2 (if ....)))
```

5.4. Piège à éviter

Souvent, en mathématique ou dans le langage courant, les définitions par cas sont données en spécifiant chacun des cas, indépendemment les uns des autres. Par exemple, pour définir la valeur absolue d'un nombre x, on écrit souvent :
– si $x \geqslant 0$, alors la valeur absolue de x est égale à x,
– si $x < 0$, alors la valeur absolue de x est égale à $-x$.

Lorsque l'on traduit une telle définition dans un langage de programmation, en particulier en Scheme, on doit – pour l'efficacité et la lisibilité – supprimer les tests inutiles. Ainsi, dans la définition Scheme que nous avons donnée, nous testons si x est supérieur ou égal à 0 et nous ne testons pas explicitement que x est inférieur à 0 (puisque, lorsque x n'est pas supérieur ou égal à 0, il est inférieur à 0).

Pour s'auto-évaluer
Exercices d'assouplissement[12]
Questions de cours[13]
Approfondissement[14]

6. Nommage de valeurs

6.1. Exemple

Nous voudrions écrire une définition de la fonction qui, étant données les longueurs des trois côtés d'un triangle quelconque, rend l'aire de ce triangle. Pour ce faire, on peut consulter un (vieux) bouquin de maths et trouver le théorème suivant :

Théorème : L'aire A d'un triangle quelconque de côtés a, b et c est telle que

$$A = \sqrt{s(s-a)(s-b)(s-c)} \ \ avec \ \ s = (a+b+c)/2$$

[12] `http://127.0.0.1:20022/q-ab-alternative-1 .quizz`
[13] `http://127.0.0.1:20022/q-ab-alternative-2 .quizz`
[14] `http://127.0.0.1:20022/q-ab-alternative-3 .quizz`

Théorème que l'on peut aussi écrire :

Théorème : Étant donné un triangle quelconque de côtés a, b et c, en nommant s la valeur $(a+b+c)/2$, son aire est égale à $\sqrt{s(s-a)(s-b)(s-c)}$.

En Scheme, nous écrivons tout simplement :

```
;;; aire-triangle : Nombre * Nombre * Nombre -> Nombre
;;; (aire-triangle a b c) rend l'aire d'un triangle de cotés «a», «b» et «c»
(define (aire-triangle a b c)
  (let ((s (/ (+ a b c ) 2)))
    (sqrt (* s (- s a) (- s b) (- s c)))))
;;; TEST pour la fontion aire-triangle (rend 6):
   (aire-triangle 3 4 5) → 6
```

et nous pourrions paraphraser le corps de cette définition en disant « nommons s la valeur de (/ (+ a b c) 2) ; la valeur rendue est (sqrt (* s (- s a) (- s b) (- s c))) ».

6.2. Grammaire

L'expression, corps de la définition de `aire-triangle` est un ***bloc*** et elle suit la règle de grammaire :

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

Ainsi une liaison « attache » à un nom (en Scheme on parle de ***variable***) une valeur, la valeur de l'expression. Une telle liaison est écrite en mettant la variable puis l'expression entre parenthèses.

Dans un bloc, on peut très bien nommer plusieurs valeurs : on aura tout simplement plusieurs liaisons. Ces différentes liaisons doivent elles-mêmes être mises entre parenthèses.

```
(let ((v1 exp1)
      (v2 exp2) )
  corps )
```

Attention, lorsque l'on n'a qu'une liaison, variable – expression sont donc entourées par deux couples de parenthèses !

```
(define (aire-triangle a b c)
  (let ((s (/ (+ a b c ) 2)))
    (sqrt (* s (- s a) (- s b) (- s c)))))
```

6.3. Exemple récapitulatif

Pour finir cette section, traitons un exemple qui utilise toutes les constructions Scheme que nous avons vues.

Le problème est d'écrire la **définition** d'une fonction qui calcule le nombre de racines d'une équation du second degré. Par exemple :

```
(nombre-racines 3 2 -3)  →  2   (Δ = 2^2 + 4×3×3 = 40)
(nombre-racines 3 2 3)   →  0   (Δ = 2^2 − 4×3×3 = −32)
(nombre-racines 1 2 1)   →  1   (Δ = 2^2 − 4×1×1 = 0)
```

L'algorithme est bien connu : étant donné une équation du second degré, ax^2+bx+c, on pose $\Delta = b^2-4ac$ et, selon que Δ est négatif, nul ou positif, le nombre de racines est respectivement 0, 1 ou 2. Voici une définition de la fonction :

```
;;; nombre-racines : Nombre * Nombre * Nombre -> nat
;;; (nombre-racines a b c) rend le nombre de racines de l'équation
;;; « a.x**2 + b.x + c = 0 »
(define (nombre-racines a b c)
  (let ((delta (- (* b b) (* 4 a c))))
    (if (< delta 0)
```

```
        (if  (= delta  0)
             1
             2))))
```

Pour s'auto-évaluer
Exercices d'assouplissement[15]
Questions de cours[16]
Approfondissement[17]

6.4. Forme `Let *`

Nous voudrions écrire une définition de la fonction qui, étant donné un nombre x rend $x^4 + x^2 + 1$. Voici une première définition :

```
;;;  f1 : Nombre -> Nombre
;;;  (f1 x) rend la valeur de « x**4 + x**2 + 1 »
(define   (f1  x)
  (+  (*  x  x  x  x)
      (*  x  x)
      1))
```

on peut vouloir nommer x^2 et x^4 :

```
;;;  f2 : Nombre -> Nombre
;;;  (f2 x) rend la valeur de « x**4 + x**2 + 1 »
(define   (f2  x)
  (let  ((x2  (*  x  x))
         (x4  (*  x  x  x  x)))
    (+  x4  x2  1)))
```

on se dit alors que l'on pourrait utiliser x^2 pour calculer x^4. Mais on ne peut pas écrire :

```
;;;  f3-ERREUR : Nombre -> Nombre
;;;  (f3-ERREUR x) rend la valeur de « x**4 + x**2 + 1 »
(define   (f3-ERREUR    x)     ; ERREUR
  (let  ((x2  (*  x  x))       ; ERREUR
         (x4  (*  x2  x2)))    ; ERREUR ERREUR ERREUR
    (+  x4  x2  1)))
```

car `x2` n'est pas connu à l'intérieur de la liste de liaisons. On peut écrire :

```
;;;  f3 : Nombre -> Nombre
;;;  (f3 x) rend la valeur de « x**4 + x**2 + 1 »
(define   (f3  x)
  (let  ((x2  (*  x  x)))
    (let  ((x4  (*  x2  x2)))
      (+  x4  x2  1))))
```

C'est lourd ! pour simplifier l'écriture, il existe une autre forme de bloc, le `let *` :

```
;;;  f4 : Nombre -> Nombre
;;;  (f4 x) rend la valeur de « x**4 + x**2 + 1 »
(define   (f4  x)
  (let *  ((x2  (*  x  x))
           (x4  (*  x2  x2)))
    (+  x4  x2  1)))
```

[15] http://127.0.0.1:20022/q-ab-nommage-1.qui zz
[16] http://127.0.0.1:20022/q-ab-nommage-2.qui zz
[17] http://127.0.0.1:20022/q-ab-nommage-3.qui zz

Ce n'est qu'une abréviation :

```
(let * ((v1 exp1) (v2 exp2) (v3 exp3))
  exp)
   ≡
(let ((v1 exp1))
  (let ((v2 exp2))
    (let ((v3 exp3))
      exp)))
```

Remarque : dans ce cours, nous aurions pu ne présenter qu'une des deux formes (`let` ou `let *`). Mais

1. Nous aurons besoin, quelques fois, de la forme `let *` et comme nous l'avons dit, son écriture sous forme de blocs imbriqués, avec des `let` , est un peu lourde.
2. Comme nous l'avons vu, on peut remplacer tout `let *` par une imbrication de `let` . L'inverse est faux. Bien plus : avec les constructions Scheme dont nous disposons, on ne peut pas remplacer un `let` par une autre construction (le `let` est donc une forme essentielle, dont on ne peut pas se passer).

Pour s'auto-évaluer
Exercices d'assouplissement[18]
Questions de cours[19]

7. Spécification d'un problème

7.1. Concepts et terminologie

Lorsqu'un donneur d'ordre (un client, un chef, un enseignant...) vous demande d'écrire un logiciel, il vous fournit un **cahier des charges**.

Par exemple, un client vous demande de pouvoir calculer l'aire d'un disque.

7.1.1. Spécification d'un problème en informatique

La spécification consiste à décrire, le plus précisément possible, la fonctionnalité du futur logiciel (ou d'un composant d'un futur logiciel) : on doit dire ce que le logiciel doit faire (le **quoi**).

Après cette phase indispensable de spécification, on devra écrire le programme (on dira que l'on **implante** le logiciel). Pour cela, on se demandera **comment** on peut faire faire à l'ordinateur la tâche attendue, celle qui est définie par la spécification.

On spécifie un problème – tout au moins pour le type de problèmes que nous traiterons dans ce cours – en répondant le plus précisément possible aux trois questions suivantes :

- Quelles sont les données ?
- Quel est le résultat ?
- Quelles propriétés relient les données et les résultats ?

Exemple :
- données : un nombre
- résultat : un nombre
- description : rend l'aire du disque de rayon le nombre donné

7.1.2. Interface, sémantique et implantation

En informatique, on a l'habitude de scinder la spécification en deux parties : l'interface et la sémantique. Prenons une métaphore pour expliquer ces deux notions : définissons un mot de la langue française qui nous était jusqu'alors inconnu, par exemple le mot « lapin ». Dans un premier temps, on peut dire que c'est un nom commun masculin. Nous pouvons alors écrire des phrases comme « le chasseur tue le lapin » ou « ce matin, un lapin a tué un chasseur ». Ce

[18] `http://127.0.0.1:20022/q-ab-let-etoile-1.` `quizz`
[19] `http://127.0.0.1:20022/q-ab-let-etoile-2.` `quizz`

Sont deux phrases qui sont parfaitement correctes en français, elles sont **syntaxiquement** correctes. Mais la seconde phrase choque celui qui connaît le **sens** du mot « lapin » : la définition du mot doit aussi indiquer quel est son sens (« Petit mammifère (lagomorphes) à longues oreilles... »).

De même, la spécification d'un composant logiciel doit être vue sous deux points de vue :

- Un point de vue syntaxique : on doit dire comment on devra l'utiliser afin d'avoir des phrases correctes par rapport à la syntaxe du langage (même si ces phrases ne veulent rien dire), autrement dit afin que le compilateur ne trouve pas d'erreur lorsqu'il compile le composant qui l'utilise. Notons que cela dépend du langage utilisé. Cette partie syntaxique de la spécification est appelée l'**interface** du composant.
- Un point de vue sémantique : le programmeur doit connaître le sens de ce nouveau « mot » afin que les logiciels écrits à l'aide de ce composant aient un sens, autrement dit qu'ils fassent bien ce que le programmeur attendaient d'eux. Cette partie sémantique de la spécification est appelée la **sémantique** du composant.

Naturellement, pour que le programme s'exécute, on doit dire aussi comment le composant logiciel doit être calculé. Un *composant logiciel* est donc un triptyque (les deux premiers points constituant la spécification) :

- son interface, c'est-à-dire la partie syntaxique du composant,
- sa sémantique, c'est-à-dire ce qu'il doit faire,
- son implantation, c'est-à-dire comment il le fait.

La solution informatique d'un problème est un programme constitué, en ce qui nous concerne, de fonctions. L'interface correspond alors à la *signature* de la fonction.

Exemple trivial :

interface (signature) : nous nommerons `aire-disque` cette fonction qui a un paramètre nombre et qui rend un nombre,

sémantique : cette fonction rend la surface du disque de rayon le nombre donné ;

implantation : une implantation triviale :

```
(define (aire-disque r)
  (* 3.1416 r r)))
```

Noter que pour ce problème trivial, il y a peu d'autres implantations. Il n'en est pas de même si l'on veut calculer l'aire d'une surface plus compliquée. En règle générale, pour un problème donné, il y a de nombreuses implantations possibles, ces implantations étant plus ou moins efficaces. Noter aussi que l'on peut très bien utiliser un composant logiciel – à partir du moment où l'on connait son interface et sa sémantique – sans connaître son implantation. Il en va ainsi de toutes les primitives du langage.

7.2. Pratiquement

7.2.1. Écriture de la spécification (convention)

Rappelons que dans nos programmes Scheme, nous écrivons la spécification des fonctions sous forme de commentaires – avec trois points-virgules – placés avant la définition. Exemple :

```
;;; aire-disque: Nombre -> Nombre
;;; (aire-disque r) rend l'aire d'un disque de rayon «r»
(* 3.1416 r r))
```

Remarquer :

- la première ligne correspond à l'interface : elle indique le nom de la fonction puis le type des données et enfin le type du résultat,
- les commentaires présents dans les lignes suivantes correspondent à la sémantique.

7.2.2. Utilisation de la spécification

Lorsqu'on utilise une fonction (*i.e.* lorsqu'on écrit une application de cette fonction), il faut

1. utiliser la signature (interface) donnée par la spécification de la fonction,
2. utiliser la sémantique donnée par la spécification de la fonction.

En ce qui concerne l'utilisation de la sémantique, il faut tout simplement considérer que la fonction rend – exactement, sans se poser d'autres questions – ce qui est dit.

Nous avons déjà dit que l'utilisation, par le programmeur de sa connaissance de la signature permettrait d'éviter des erreurs de compilation (pas de faute d'orthographe dans le nom de la fonction, application ayant un bon nombre d'arguments). Mais il faut aussi utiliser la connaissance, présente dans la signature, des types des données et du résultat en effectuant une *vérification de type*.

7.2.3. Vérification de type

Rappelons qu'un programme Scheme est une suite de définitions de fonctions et d'expressions, l'interprète affichant les résultats de l'évaluation des expressions dans l'environnement qui contient les définitions.

Notons que dans un programme réel, il y a beaucoup de définitions de fonctions et une seule expression : nous n'effectuerons la vérification de type que pour les définitions de fonctions.

Définition de fonction

Pour que la définition

;;; f : α * β -> γ
```
(define  (f a b) exp)
```

soit correcte vis-à-vis du typage, il faut que

- le nombre de variables de la fonction soit égal au nombre de types des donnés de la signature,
- le type de l'expression `exp` soit γ (le type qui a été donné comme type du résultat dans la signature) lorsque le type de `a` (resp. `b`) est α (resp. β) (les types qui ont été donnés comme types des données dans la signature) .

Ainsi, pour vérifier qu'une définition de fonction est correcte vis-à-vis des types, la question est de savoir si une expression (au départ l'expression de la définition)

- est bien d'un certain type (au départ le type du résultat de la fonction)
- sachant que les variables (les variables de la fonction) sont d'un certain type (le type des données de la fonction).

Expressions présentes dans une définition de fonction

Dans cette étude nous ne traiterons que deux sortes d'expressions, les applications et les alternatives, et la vérification consiste donc à vérifier qu'une expression est de type δ :

a) lorsque l'expression est une application, elle est de la forme `(g e1 e2)` et pour qu'elle soit de type δ :
   - il faut que le nombre d'arguments de l'application soit égal au nombre de types des donnés de `g`,
   - il faut que le type du résultat de `g` (type qui est donné par la signature de `g`) soit δ,
   - si le type des données de `g` est $\alpha_1 * \beta_1$ (type qui est donné par la signature de `g`), il faut que l'expression `e1` (resp. `e2`) soit de type α_1 (resp. β_1).

b) lorsque l'expression est une alternative, elle est de la forme `(if c e1 e2)` et pour qu'elle soit de type δ, il faut que
   - `c` soit de type `bool` (prédicat),
   - les expressions `e1` et `e2` soient de type δ.

Exemple : considérons la définition suivante :

```
;;; max: Nombre * Nombre -> Nombre
(define  (max m n)
  (if  (< m n) n m))
```

pour vérifier qu'elle est bien typée, il faut :

1. vérifier l'en-tête de la définition de la fonction...
2. vérifier le type de l'expression (qui doit être `Nombre` lorsque `m` et `n` sont de type `Nombre`) :
   c'est une alternative...
   (a) la condition est une application (qui doit être de type `bool` ou η + #f)...
   (b) la conséquence est la variable `n` qui est bien, par hypothèse, de type `Nombre` ,
   (c) l'alternant est la variable `m` qui est bien, par hypothèse, de type `Nombre` .

7.2.4. Recommandation très importante

Toutes les fois que vous écrivez une définition de fonction, vous devez vérifier son typage, cela évitera un très grand nombre d'erreurs.

7.3. À propos des erreurs

7.3.1. Taxinomie des erreurs

On peut déjà classer les erreurs en deux catégories :

- les erreurs détectées par l'évaluateur,
- les erreurs non détectées par l'évaluateur.

Clairement, les erreurs qui posent le plus de problèmes sont les erreurs qui ne sont pas détectées : le programme affiche (ou imprime) un résultat, qui a l'air d'un résultat, et l'utilisateur s'en sert comme tel, mais qui n'est pas le bon résultat. Le plus souvent ce sont des erreurs de conception du programme (le remède étant de programmer avec méthode, en particulier de toujours vérifier le typage des définitions de fonctions), mais cela peut être dû aussi à une mauvaise utilisation d'un logiciel mal écrit (nous reviendrons sur ce point dans un instant).

Classons maintenant les erreurs détectées par l'évaluateur :

Erreurs détectées par l'évaluateur

- lors de l'analyse de l'expression
  - erreurs de syntaxe
    ```
    (define (f x))
    define: malformed definition
    ```
  - nom de fonction ou variable inconnue
    ```
    (sqt 4)
    reference to undefined identifier: sqt
    ```
- lors de l'évaluation de l'expression
  - erreur de type
    ```
    (sqrt "toto")
    sqrt: expects argument of type <number>; given "toto"
    ```
  - résultat non défini
    ```
    (/ 1 0)
    /: division by zero
    ```

7.3.2. Erreurs et spécification

Lorsque nous écrivons « `;;; aire-disque: Nombre -> Nombre` », nous indiquons à l'utilisateur de cette fonction qu'il faut que la donnée de cette fonction soit un nombre. Lorsqu'il écrit une application de cette fonction, un utilisateur doit alors obligatoirement vérifier que les arguments sont bien des valeurs numériques, sous peine d'avoir une erreur détectée lors de l'évaluation, voire, pire, un résultat sans signification.

On peut aussi préciser le domaine de validité, ce que nous faisons en écrivant la contrainte derrière le type et entre /. Par exemple, le rayon d'un disque doit être positif ou nul :

`;;;` *aire-disque: Nombre/>=0/ -> Nombre*

Considérons maintenant le problème du calcul de l'aire d'une couronne. La donnée est constituée par deux nombres positifs, le résultat est un nombre (positif) égal à l'aire de la couronne de rayon extérieur le premier nombre donné et de rayon intérieur le second nombre donné. Mais la signature :

`;;;` *aire-couronne: Nombre/>=0/ * Nombre/>=0/ -> Nombre*

n'est pas complètement satisfaisante. En effet, pour que les données aient un sens, il faut de plus que le rayon extérieur soit supérieur ou égal au rayon intérieur. Que fait-on dans le cas contraire ? Deux solutions :

- l'implantation de la fonction détecte l'erreur,
- l'implantation de la fonction ne détecte pas l'erreur.

Voyons tout d'abord la première implantation (la seconde est triviale) :

```
(define (aire-couronne r1 r2)
  (if (< r1 r2)
      (erreur 'aire-couronne
              "rayon extérieur (" r1 ") <"
              "rayon intérieur (" r2 ")")
      (- (aire-disque r1) (aire-disque r2))))
```

et un exemple d'application :

```
(aire-couronne     5 7)
```
aire-couronne : ERREUR : rayon extérieur (5) < rayon intérieur (7)

Noter la fonction (`erreur`) utilisée pour signifier l'erreur :
- elle a un nombre quelconque d'arguments, le premier argument étant, par convention, le nom de la fonction où est détectée l'erreur précédé d'une apostrophe ;
- elle affiche le nom de la fonction, puis deux points, puis « ERREUR » et enfin les autres arguments ;
- en plus, son évaluation termine toute l'évaluation en cours.

Nous disposons aussi d'une fonction (`erreur?`) qui teste si une fonction, appliquée à des arguments donnés, provoque une erreur. Par exemple :

```
(erreur?   aire-couronne    5 3) →#F
(erreur?   aire-couronne    5 7) →#T
```

Pour signifier à l'utilisateur que cette fonction signifie une erreur lorsque r1 est inférieur à r2, nous l'indiquons après la signature :

```
;;; aire-couronne : Nombre/>=0/ * Nombre/>=0/ -> Nombre
;;; (aire-couronne r1 r2) rend l'aire de la couronne de rayon extérieur «r1» et de rayon intérieur «r2»
;;; ERREUR lorsque r1 < r2
(define   (aire-couronne     r1  r2)
  (if  (<  r1  r2)
       (erreur   'aire-couronne
                 "rayon   extérieur    ("  r1  ")  <"
                 "rayon   intérieur    ("  r2  ")")
       (-  (aire-disque     r1)  (aire-disque     r2))))
```

Noter que la détection de l'erreur a un certain coût en temps alors qu'il se peut que, lors de l'utilisation de la fonction, nous soyons sûrs qu'elle est utilisée dans de bonnes conditions (parce que les arguments sont calculés par programme, programme tel que...). Il est alors dommage de perdre du temps d'ordinateur pour rien. Dans ce cas, l'implantation ne fait pas la vérification. Il est alors indispensable — car on est dans le cas où le programme rend un résultat, celui-ci n'ayant pas de sens — de l'indiquer aux utilisateurs de cette fonction :

```
;;; aire-couronne-sans : Nombre/>=0/ * Nombre/>=0/ -> Nombre
;;; (aire-couronne-sans r1 r2) rend l'aire de la couronne de rayon extérieur
;;; «r1» et de rayon intérieur «r2»
;;; HYPOTHÈSE : r1 >= r2
(define   (aire-couronne-sans       r1  r2)
  (-  (aire-disque     r1)  (aire-disque     r2)))
```

En résumé, lorsqu'on utilise une fonction, on doit suivre la spécification, sachant que

- il faut que les arguments soient du type précisé (et, *a priori*, on ne connait pas le comportement de la fonction dans le cas contraire) ;
- il faut que les arguments vérifient les hypothèses et ne vérifient pas les cas d'erreurs, sachant que
  - une erreur est signalée dans le second cas,
  - aucune erreur n'est signalée dans le premier cas.

Pour s'auto-évaluer
Exercices d'assouplissement[20]
Questions de cours[21]

[20] `http://127.0.0.1:20022/q-ab-specification -1.quiz z`
[21] `http://127.0.0.1:20022/q-ab-specification -2.quiz z`

8. Écriture d'algorithmes récursifs

8.1. Compréhension de la récursivité

Considérons la définition suivante :

```
;;; ! : nat/>0/ -> nat
;;; (! n) rend la factorielle de «n»
(define (! n)
  (if (= n 1)
      1
      (* n (! (- n 1)))))
```

Une telle définition (où la fonction à définir est utilisée dans le corps de la définition) est une **définition récursive**.
Évaluons, en pas à pas, `(! 3)`

```
(! 3)
(if (= 3 1) 1 (* 3 (! (- 3 1))))
(if false 1 (* 3 (! (- 3 1))))
(* 3 (! (- 3 1)))
(* 3 (! 2))
(* 3 (if (= 2 1) 1 (* 2 (! (- 2 1)))))
(* 3 (if false 1 (* 2 (! (- 2 1)))))
(* 3 (* 2 (! (- 2 1))))
(* 3 (* 2 (! 1)))
(* 3 (* 2 (if (= 1 1) 1 (* 1 (! (- 1 1))))))
(* 3 (* 2 (if true 1 (* 1 (! (- 1 1))))))
(* 3 (* 2 1))
(* 3 2)
6
```

Ainsi, l'évaluation de `(! 3)` rend 6 qui est bien égal à factorielle de 3.

Pourquoi « ça marche » ?

a) lorsque n n'est pas égal à 1, $n!$ est égal à $n \times (n-1)!$
(nécessaire car, intuitivement, dans le calcul précédent, nous disons que 3! est égal à $3 \times (3-1)!$ et que 2! est égal à $2 \times (2-1)!$ puisque nous remplaçons 3! par $3 \times (3-1)!$ et 2! par $2 \times (2-1)!$),

b) 1! est égal à 1 (nécessaire car, intuitivement, dans le calcul précédent, nous disons que 1! est égal à 1 puisque nous remplaçons 1! par 1),

c) $n-1$ est strictement plus petit que n.
Par exemple, si l'on n'imposait pas cette condition, on pourrait donner comme définition de factorielle :

```
(define (! n)
    (/ (! (+ n 1)) (+ n 1)))
```

mais alors le calcul serait

```
(! 2)
(/ (! (+ 2 1)) (+ 2 1))
```

```
(/ (! 3) (+ 2 1))
(/ (/ (! (+ 3 1)) (+ 3 1)) (+ 2 1))
(/ (/ (! 4) (+ 3 1)) (+ 2 1))
```

...

et on pourrait attendre longtemps le résultat !

8.2. Écriture d'algorithmes récursifs

Dans cette section, nous étudions comment on peut écrire, en général, une définition récursive. Pour ce faire, nous allons donner deux exemples d'écritures de définitions récursives puis nous synthétiserons une méthode de travail.

En fait, les deux exemples sont deux algorithmes différents de la même fonction, la fonction qui, à deux entiers naturels n et m, associe l'entier naturel n^m.

8.2.1. Premier algorithme

Idée de départ : n^m est égal à $n \times n^{m-1}$. Mais peut-on dire que $n^m = n \times n^{m-1}$ (sous-entendu pour tout entier naturel n et tout entier naturel m) ? Non car l'égalité est fausse lorsque m est égal à 0 (-1 n'appartient pas aux entiers naturels !). Nous devons donc « enlever » de l'appel récursif le cas où m est nul. Pour ce faire nous utilisons une alternative `(if (= m 0) ...)` et, pour écrire le conséquent, nous remarquons que n^0 est égal à 1. La définition est donc

```
;;; ^ : nat * nat -> nat
;;; (^ n m) rend n^m
(define (^ n m)
  (if (= m 0)
      1
      (* n (^ n (- m 1)))))
```

Remarque : avec cet algorithme, l'évaluation de n^m est effectué avec m multiplications.

8.2.2. Second algorithme

Idée de départ :

- lorsque m est pair, n^m est égal à $(n^{m\div 2})^2$,
- lorsque m est impair, n^m est égal à $n \times (n^{m\div 2})^2$

On a ainsi des égalités où les futurs appels récursifs sont effectués sur des valeurs plus petites ($m \div 2$ est plus petit que m). Est-ce que ces égalités sont toujours vraies ? Oui. Peut-on écrire

```
(define (^ n m)
  (if (even? m)
      (carre (^ n (quotient m 2)))
      (* n (carre (^ n (quotient m 2))))))
```

comme définition ? Non ! car pour m nul, $m \div 2$ est égal à 0 soit à m et n'est donc pas **strictement** plus petit que m : il faut encore « enlever » 0 du cas général. La bonne définition est donc :

```
;;; ^ : nat * nat -> nat
;;; (^ n m) rend n^m
(define (^ n m)
  (if (= m 0)
      1
      (if (even? m)
          (carre (^ n (quotient m 2)))
          (* n (carre (^ n (quotient m 2)))))))
```

avec la fonction `carre` spécifiée par :

```
;;; carre : int -> nat
;;; (carre n) rend le carré de n
```

et que vous pouvez implanter en exercice.

Remarque : avec cet algorithme, lors de l'évaluation de n^m le nombre de multiplications est égal au nombre de fois que l'on peut effectuer l'opération diviser m par 2, puis le résultat obtenu par 2, puis le résultat obtenu par 2... jusqu'à trouver 0 : il est, grosso modo, égal au logarithme en base 2 de m, nous dirons qu'il est de l'ordre de $lg(n)$.

Terminologie : dans l'appel récursif, on divise par deux une donnée : on dit que l'on travaille par **dichotomie**.

8.2.3. Méthode de travail

Supposons que l'on veuille implanter la fonction

$F : X \rightarrow Y$

$\| \; F(x)$ est égal à $f(x)$

Pour écrire une défi nition récursive, on doit suivre la méthode suivante :

a) Décomposer la donnée (x étant la donnée, $d_1(x)$... sont des éléments de X « plus petits » que x).

b) Écrire la relation de récurrence, c'est-à-dire une égalité qui est vraie (presque toujours) entre $f(x)$ et une expression (dite de récurrence) qui contient des occurrences de $f(d_i(x))$.

c) Déterminer les valeurs (« de base ») pour lesquelles :

   i) l'égalité est fausse ; en particulier, regarder quand l'expression de récurrence n'est pas défi nie ;

   ii) les valeurs $d_i(x)$ ne sont pas **strictement** « plus petites » que x ; en particulier, regarder si $d_i(x)$ peut être égal à x.

d) Déterminer la valeur de la fonction f pour les valeurs de base.

Remarque : Dans les explications précédentes, « f » est une fonction connue, c'est-à-dire que l'on peut, pour toute valeur x de X dire quelle est la valeur de $f(x)$. En général, nous nommerons de la même façon la fonction à implanter. Ici, nous réservons le « f » minuscule pour la spécifi cation et le « F » majuscule pour le nom de la fonction que l'on implante afi n de bien voir que l'on raisonne sur la spécifi cation et non sur l'implantation.

Pour s'auto-évaluer
Exercices d'assouplissement[22]
Questions de cours[23]

9. Notion de liste

Vous avez écrit un programme pour calculer la moyenne de trois notes. Il est naturel de se demander comment faire pour calculer la moyenne de n notes. C'est ce que nous allons voir maintenant.

9.1. Deux notions importantes

9.1.1. Notion de séquence en informatique

Considérons le problème de la moyenne de n notes. La donnée d'un tel problème est une **séquence fi nie** de notes (en particulier s'il y a des coeffi cients).

Une telle séquence peut être vue de différentes façons :

- de façon complètement informelle comme la donnée de n_0, n_1,...,n_{p-1} ;
- ce que l'on peut aussi noter $n(0)$, $n(1)$,... $n(p-1)$: c'est une application de $\{0,1,2,...p-1\}$ dans 0..20 (nous verrons plus tard cette vision des séquences) ;
- mais on peut aussi la voir, d'une façon récursive, comme étant constituée par :
  - la première note,
  - et les autres notes qui constituent encore une séquence de notes (où il y a moins de notes que dans la séquence complète).

  Récursivement, on arrive alors à une séquence qui n'a qu'un élément et, pourquoi ne pas continuer, on arrive à la séquence qui n'a pas d'élément et que l'on nomme la **séquence vide**.

[22] `http://127.0.0.1:20022/q-ab-recursivite-n -1.quiz z`
[23] `http://127.0.0.1:20022/q-ab-recursivite-n -2.quiz z`

En Scheme, cette dernière vision des séquences est mise en œuvre par la notion de liste (cf. la section suivante).

9.1.2. Notion de structures de données

Jusqu'à présent, les données et les résultats des fonctions que nous avons étudiées étaient des entiers, des réels ou des booléens. Il s'agissait d'informations simples, même lorsque nous avons écrit une fonction qui calcule la moyenne de trois notes où nous manipulions trois informations simples, représentées par trois variables dans la définition de la fonction.

Pour calculer la moyenne de *n* notes, on ne peut pas écrire la définition d'une fonction qui aurait *n* variables représentant les *n* notes données : il faut que la donnée des *n* notes soit **une** (unique) donnée de la fonction. Mais alors cette donnée est constituée de plusieurs informations et, pour retrouver chacune d'elles, on doit structurer la donnée dans ce qu'on nomme une **structure de données**.

Encore faut-il pouvoir
- fabriquer une telle donnée structurée,
- retrouver les différentes informations présentes dans la donnée structurée.

Pour ce faire, on peut utiliser des fonctions que l'on nomme des **fonctions de base** :
- un **constructeur** (le type du résultat d'une telle fonction est la structure de données) permet de fabriquer une donnée structurée, éventuellement à partir d'une valeur structurée ;
- un **accesseur** (le type de la donnée d'une telle fonction est la structure de données) permet de retrouver une information présente dans une donnée structurée, sous réserve que la donnée contienne une telle information ;
- un **reconnaisseur** (le type de la donnée d'une telle fonction est la structure de données et son résultat est un booléen) permet de savoir quelle est la forme d'une donnée structurée, essentiellement pour savoir si on peut lui appliquer un accesseur.

9.2. Structure de données «liste»

Une liste est une structure de données qui regroupe une séquence d'éléments de même type. Par exemple, la donnée de la fonction qui calcule la moyenne de *n* notes est une liste de nombres naturels.

Nous noterons `LISTE[Note]` (`Note` étant le type `nat/<=20/`) un tel type et la signature de la fonction est donc :

```
;;; moyenne-notes : LISTE[Note] -> Note
;;;   avec Note = nat/<=20/
```

D'une façon générale, nous noterons `alpha`, `beta` ... ou α, β... un type quelconque (autrement dit, on pourra remplacer ces noms de type par n'importe quel type) et `LISTE[alpha]` une liste d'éléments de type `alpha`, `LISTE[beta]` une liste d'éléments de type `beta` ... `LISTE[`α`]` une liste d'éléments de type α, `LISTE[`β`]` une liste d'éléments de type β...

Étudions maintenant les fonctions de base sur les listes (à savoir les constructeurs `list` et `cons`, les accesseurs `car` et `cdr` et le reconnaisseur `pair?`).

9.2.1. Constructeurs

Pour construire une liste, on utilise la fonction `list` :

```
;;; list: alpha *... -> LISTE[alpha]
;;; (list v...) rend une liste dont les termes sont les arguments
;;; (list) rend la liste vide
```

Par exemple,

```
(list 3 5 2 5) → (3 5 2 5)
(list) → ()
```

On peut aussi construire une liste, à partir d'une autre liste, en utilisant la fonction `cons` :

```
;;; cons: alpha * LISTE[alpha] -> LISTE[alpha]
;;; (cons v L) rend la liste dont le premier élément est «v» et dont les
;;; éléments suivants sont les éléments de la liste «L».
```

Par exemple :

```
(cons 7 (list 3 5 2 5)) → (7 3 5 2 5)
(cons 5 (cons 2 (cons 5 (list)))) → (5 2 5)
```

Noter la signature de la fonction cons :

1. elle a deux données
   (a) un élément de n'importe quoi (d'où le premier `alpha`),
   (b) une liste (d'où `LISTE[...]`) de ce même n'importe quoi (d'où le deuxième `alpha`) ;
2. elle rend une liste (d'où le second `LISTE[ ]`) de ce même n'importe quoi (d'où le troisième `alpha`).

On peut aussi le dire autrement : dans la pratique, on a un certain type, nommons le `alpha` , et on veut construire une liste d'éléments de type `alpha` . Pour ce faire, on peut utiliser la fonction `cons` , avec comme premier argument un élément de type `alpha` et comme second argument une liste de type `LISTE[alpha]` , le résultat étant de type `LISTE[alpha]` .

Noter aussi la signature de la fonction `list` : c'est une fonction qui a un nombre quelconque d'arguments, tous du même type (d'où les points de suspension), le type étant quelconque (d'où le premier `alpha`) et elle rend une liste (d'où le `LISTE[ ]`) de ce même n'importe quoi (d'où le deuxième `alpha`).
Remarque : en fait, on n'a pas besoin du constructeur `list` , sauf pour construire la liste vide. En effet, par exemple,
`(list 5 2 3)` ≡ `(cons 5 (cons 2 (cons 3 (list))))`

9.2.2. Accesseurs

Le premier élément d'une liste non vide est donné par la fonction `car` :

```
;;; car: LISTE[alpha] -> alpha
;;; ERREUR lorsque la liste donnée est vide
;;; (car L) rend le premier élément de la liste «L».
```

Par exemple :

`(car (list 3 5 2 5))` → `3`

Le reste d'une liste non vide est donné par la fonction `cdr` :

```
;;; cdr: LISTE[alpha] -> LISTE[alpha]
;;; ERREUR lorsque la liste donnée est vide
;;; (cdr L) rend la liste des termes de «L» sauf son premier élément.
```

Par exemple :

`(cdr (list 3 5 2 5))` → `(5 2 5)`

Remarques :

– pour toute liste *l* et toute valeur *x*, on a
  i) `(car (cons x l))` ≡ `x`
  ii) `(cdr (cons x l))` ≡ `l`
  qui est une propriété caractéristique des listes.
– pour toute liste non vide `l`, on a
  `l` ≡ `(cons (car l) (cdr l))` .

9.2.3. Reconnaisseur

Il faut aussi savoir si une liste n'est pas vide : ceci est possible grâce au prédicat `pair?` :

```
;;; pair?: LISTE[alpha] -> bool
;;; (pair? L) rend vrai ssi la liste «L» n'est pas vide.
```

Par exemple :

```
(pair? (list 3 5 2 5)) → #T
(pair? (list))         → #F
```

9.3. Exemples de définitions simples sur les listes

Exemple 1 : pour avoir le deuxième élément d'une liste donnée :

```
;;; deuxieme : LISTE[alpha] -> alpha
```

```
;;; ERREUR lorsque la liste donnée a moins de deux éléments
;;; (deuxieme L) rend le deuxième élément de la liste «L».
(define (deuxieme L)
  (car (cdr L)))
  ;;; Essai :
  (deuxieme (list 3 5 2 5)) → 5
```

Exemple 2 : pour avoir le reste après le deuxième élément d'une liste donnée :

```
;;; reste2 : LISTE[alpha] -> LISTE[alpha]
;;; ERREUR lorsque la liste donnée a moins de deux éléments
;;; rend la liste des termes de «L» sauf ses deux premiers éléments.
(define (reste2 L)
  (cdr (cdr L)))
  ;;; Essai :
  (reste2 (list 3 5 2 5)) → (2 5)
```

Exemple 3 : prédicat `lg>1?` pour déterminer si une liste donnée a au moins 2 éléments (par exemple `(lg>1? (list 3)) -> false` et `(lg>1? (list 3 4)) -> true`.

```
;;; lg>1? : LISTE[alpha] -> bool
;;; (lg>1? L) rend vrai ssi la liste «L» a au moins 2 éléments
(define (lg>1? L)
  (and (pair? L)
       (pair? (cdr L))))
```

Essais :

```
(lg>1? (list 3 5 2 5)) → #t
(lg>1? (list)) → #f
(lg>1? (list 3)) → #f
(lg>1? (list 3 5)) → #t
```

Rappel : `and` est une forme spéciale (si le premier argument est faux, on n'évalue pas le second) ; indispensable ici car, lorsque la liste donnée est vide, `(cdr L)` donne une erreur.

9.3.1. Abréviations

<Abréviations> → `cad*r`
`cd*r`

En fait on devra souvent désigner le deuxième, troisième... élément d'une liste ou ce qui reste après le deuxième, troisième... élément d'une liste. Aussi Scheme a des abréviations qui permettent de désigner ces valeurs. La règle de formation de ces abréviations est très simple : on écrit « c », puis, éventuellement, un « a » puis des « d » et enfi n un « r ». Par exemple, `caddr` est une telle abréviation. Une telle abréviation remplace une expression combinant des `car` et des `cdr`, le « a » correspondant à un `car` et un « d » correspondant à un `cdr`.

Par exemple

Exemple : `(caddr L)` ≡ `(car (cdr (cdr L)))`

`(caddr L)` est une abréviation pour `(car (cdr (cdr L)))`.

Remarque : on verra plus tard que l'on peut mettre d'autres `a`.

Pour s'auto-évaluer
Exercices d'assouplissement[24]

Pour s'auto-évaluer
Exercices d'assouplissement[25]

Pour s'auto-évaluer

[24] `http://127.0.0.1:20022/q-ab-liste-1-1.qui` zz
[25] `http://127.0.0.1:20022/q-ab-liste-2-1.qui` zz

10. Définitions récursives sur les listes

10.1. Premier exemple d'une définition récursive sur les listes

Écrivons une définition de la fonction `longueur` qui rend la longueur d'une liste donnée :

```
(longueur (list 3 5 2 5)) → 4
(longueur (list)) → 0
(longueur (list 3)) → 1
(longueur (list 3 5)) → 2
```

– lorsque la liste n'est pas vide, sa longueur est égale à 1 plus la longueur de la liste « qui reste » ;
– la longueur de la liste vide est 0.

D'où le source Scheme :

```
;;; longueur : LISTE[alpha] -> nat
;;; (longueur L) rend la longueur de la liste «L» donnée
(define (longueur L)
  (if (pair? L)
      (+ 1 (longueur (cdr L)))
      0))
```

Dans la définition précédente, pour définir la fonction `longueur`, nous utilisons cette même fonction : cette définition est donc récursive.

Vous pouvez regarder comment DrScheme évalue `(longueur (list 3 5))` en lui demandant d'évaluer cette expression en « pas à pas » (icone « Step »).

En utilisant la fonction `longueur`, on peut donner une nouvelle définition de la fonction `lg>1?` vue précédemment :

```
;;; lg>1? : LISTE[alpha] -> bool
;;; (lg>1? L) rend vrai ssi la liste «L» a au moins 2 éléments
(define (lg>1? L)
  (> (longueur L) 1))
```

Cette définition paraît meilleure (elle est plus simple) que la définition donnée précédemment. **Elle est à proscrire** car elle est beaucoup moins efficace :

– avec la première définition, une évaluation de `(lg>1? L)` applique la fonction `pair?` au plus deux fois,
– avec la seconde définition, si n est la longueur de la liste `L`, l'évaluation de `(lg>1? L)` applique la fonction `pair?` $n+1$ fois.

Remarque : de fait cette fonction est une primitive de Scheme (et elle se nomme `length`). Dans la suite du cours, nous redéfinirons d'autres primitives (car ce sont des exemples intéressants et parce que, pour évaluer l'efficacité des algorithmes, il faut savoir comment elles sont implantées), mais en les nommant alors avec le mot anglais (nous n'avons pas pu le faire ici car nous voulions faire du pas à pas).

10.2. Schéma de récursion (simple) sur les listes

La définition de la notion de liste étant récursive, il n'est pas étonnant que la plupart des définitions de fonctions sur les listes soient des définitions récursives. Le schéma récursif fondamental sur les listes est semblable à la définition récursive des listes : on commence par traiter le cas général d'une liste non vide (appel récursif sur le reste de la liste et combinaison du résultat avec le premier de ses éléments), puis l'on traite le cas de base (la liste vide).

[26] `http://127.0.0.1:20022/q-ab-liste-3-1.qui` zz
[27] `http://127.0.0.1:20022/q-ab-liste-3-2.qui` zz
[28] `http://127.0.0.1:20022/q-ab-liste-3-3.qui` zz

```
;;; fonction: LISTE[alpha] -> ...
(define  (fonction L)
  (if  (pair?  L)
       (combinaison (car  L)
                    (fonction (cdr  L)))
       base ) )
```

où *base* est la valeur à rendre lorsque la liste est vide, et *combinaison* est la fonction combinant le résultat de l'appel récursif avec le premier élément de la liste. Ainsi, pour la fonction longueur (la combinaison n'utilise pas (car L)) :

```
;;;  longueur : LISTE[alpha] -> nat
;;;  (longueur L) rend la longueur de la liste «L» donnée
(define  (longueur  L)
  (if  (pair?  L)
       (+ 1 (longueur  (cdr  L)))
       0))
```

Schéma	longueur
base	0
(*combinaison* (car L) (*fonction* (cdr L)))	(+ 1 (longueur (cdr L)))

10.2.1. Exemples de définitions récursives

Exemple 1 : écrivons une définition de la fonction `somme` qui rend la somme des éléments d'une liste de nombres donnée :

```
(somme  (list  2 5 7))  → 14
(somme  (list))  → 0
```

- lorsque la liste donnée n'est pas vide, la somme de ses éléments est égale à la valeur de son premier élément plus la somme des éléments du cdr de la liste (*combinaison* $\equiv +$) ;
- la somme des éléments d'une liste vide est 0 (*base* $\equiv 0$) :

```
;;;  somme : LISTE[Nombre] -> Nombre
;;;  (somme L) rend la somme des éléments de la liste «L»
;;;  (rend 0 pour la liste vide)
(define  (somme  L)
  (if  (pair?  L)
       (+ (car  L) (somme  (cdr  L)))
       0))
```

Exemple 2 : écrivons une définition de la fonction `conc-d` qui ajoute un élément à la fin d'une liste :

```
(conc-d  (list  1 2 3) 4)  → (1 2 3 4)
(conc-d  (list)  1)  → (1)
```

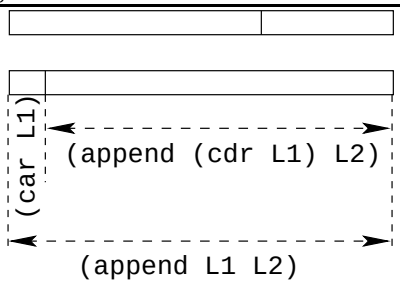
- *combinaison* $\equiv$ `cons` ,
- *base* $\equiv$ `(list x)` :

```
;;;  conc-d : LISTE[alpha] * alpha -> LISTE[alpha]
;;;  (conc-d L x) rend la liste obtenue en ajoutant «x» à la fin de la liste «L».
(define  (conc-d  L x)
  (if  (pair?  L)
       (cons  (car  L) (conc-d  (cdr  L) x))
       (list  x)))
```

Exemple 3 : écrivons une définition de la fonction `append` qui concatène (met « bout à bout ») deux listes données. Par exemple :

```
(append  (list  1 2) (list  3 4 5))  → (1 2 3 4 5)
```

Il suffit de faire une récursion sur la première liste comme le montre le dessin suivant :



– *combinaison* ≡ cons ,
– *base* ≡ L2 :

```
;;; append : LISTE[alpha] * LISTE[alpha] -> LISTE[alpha]
;;; (append L1 L2) rend la concaténation de «L1» et de «L2»
(define (append L1 L2)
  (if (pair? L1)
      (cons (car L1)
            (append (cdr L1) L2))
      L2))
```

10.3. Retour sur la méthode de travail pour écrire des définitions récursives

10.3.1. Exemple

Nous voudrions écrire la fonction eme telle que

(eme 3 (list 1 2 5 3 4 3 4)) → 4
(eme 3 (list 1 2)) → 0

(eme a L) rend l'entier *p* défini comme suit : si a est égal au premier élément de la liste L, *p* est égal à 1, sinon, si a est égal au deuxième élément de liste, *p* est égal à 2... et si a n'a pas d'occurrence dans L, *p* est égal à 0.

Idée : en général, si *a* est égal à $(carL)$, $eme(a,L)$ est égal à 1 et, sinon, $eme(a,L)$ est égal à $1+eme(a,(cdrL))$ et il faut « enlever » la liste vide :

```
(define (eme-faux a L)                    ; erroné
  (if (pair? L)                           ; erroné
      (if (= (car L) a)                   ; erroné
          1                               ; erroné
          (+ 1 (eme-faux a (cdr L))))     ; erroné
      0)                                  ; erroné
 )
```

Bien sûr, vu ce qui est écrit, le programme est faux. En effet, si on l'essaie :

(eme-faux 3 (list 1 2 5 3 4 3 4)) → 4
(eme-faux 3 (list 1 2)) → 2

Que c'est-il passé ?

En fait, dans la spécification, le cas où *a* n'a pas d'occurrence dans *L* est un cas particulier et, dans ce cas, notre égalité est fausse (encore une « démonstration » de $1+0=0$!) et nous devons donc le traiter comme un cas particulier. Comment reconnait-on ce cas particulier ? c'est que le résultat est nul :

```
(define (eme-0 a L)
  (if (pair? L)
      (if (= (car L) a)
          1
          (if (= 0 (eme-0 a (cdr L)))
              0
              (+ 1 (eme-0 a (cdr L)))))
```

0))

10.3.2. Méthode de travail

Nous rappelons la méthode de travail pour l'écriture de définitions récursives en ajoutant, par rapport à ce qui a été donné précédemment, le problème des cas particuliers dans la spécification.

Pour écrire une définition récursive, on doit suivre la méthode suivante :

a) Décomposer la donnée (x étant la donnée, $d_1(x)$... sont des éléments de X « plus petits » que x).

b) Écrire la relation de récurrence, c'est-à-dire une égalité qui est vraie (presque toujours) entre $f(x)$ et une expression (dite de récurrence) qui contient des occurrences de $f(d_i(x))$.

c) Déterminer les valeurs (« de base ») pour lesquelles :

   i) l'égalité est fausse ; en particulier,
      – regarder quand l'expression de récurrence n'est pas définie et
      – regarder les valeurs (de x et de $d_i(x)$) où l'une des fonctions utilisées (la fonction à implanter et les autres fonctions intervenant dans l'expression de récurrence) est définie par un cas particulier ;

   ii) les valeurs $d_i(x)$ ne sont pas **strictement** « plus petites » que x ; en particulier, regarder si $d_i(x)$ peut être égal à x.

d) Déterminer la valeur de la fonction f pour les valeurs de base.

Remarque : lorsque la spécification d'une fonction comporte des cas particuliers, il faut en tenir compte, non seulement lors de l'implantation de la fonction, mais aussi lors de toute implantation – de n'importe quelle fonction – qui utilise cette fonction (cf. le cas i) du cas c) de la méthode donnée ci-dessus). Bien évidemment cela est une source importante d'erreurs : il faut éviter le plus possible d'avoir des cas particuliers dans la spécification des fonctions !

10.3.3. Efficacité et nommage de valeurs

La définition donnée pour la fonction `eme-0` est très, très mauvaise. Elle est juste (la fonction rend bien ce que l'on attend d'elle, mais elle est inacceptable ! En effet, dans l'expression définissant la fonction, nous calculons deux fois `(eme-0 a (cdr L))`. Dans notre idée, il suffisait de parcourir tous les éléments de la liste, le temps est donc de l'ordre de n, si n est le nombre d'éléments de la liste donnée. Or, avec la définition que nous avons donné, pour calculer `(eme-0 L)`, il faut donc un peu plus que deux fois le temps pour calculer `(eme-0 a (cdr L))`. Si `L` a n éléments, `(cdr L)` a $n-1$ éléments : en nommant t_n le temps de calcul de l'application de la fonction à une liste de n éléments, t_n est plus grand que $2 \times t_{n-1}$. t_n est donc au moins de l'ordre de 2^n : dès que la liste sera un peu longue, le temps de calcul sera prohébitif et personne n'aura le courage d'attendre le résultat !

Oui, mais, on a besoin à deux endroits de la valeur de `(eme-0 a (cdr L))`. Pour n'évaluer qu'une fois cette application, il faut nommer sa valeur (à l'aide d'un `let`) :

```
;;; eme : alpha * LISTE[alpha] -> nat
;;; (eme a L) rend l'entier «p» défini comme suit : si «a» est égal au premier
;;; élément de la liste «L», «p» est égal à 1, sinon, si «a» est égal au
;;; deuxième élément de la liste «L», «p» est égal à 2... et si «a» n'a pas
;;; d'occurrence dans «L», «p» est égal à 0.
(define (eme a L)
  (if (pair? L)
      (if (= (car L) a)
          1
          (let ((eme-a-cdr-L (eme a (cdr L))))
            (if (= 0 eme-a-cdr-L)
                0
                (+ 1 eme-a-cdr-L))))
      0))
```

Noter que l'on doit écrire le bloc dans l'alternant de l'alternative car, avant, `(cdr L)` peut ne pas être définie (lorsque `L` est vide).

Pour s'auto-évaluer
Exercices d'assouplissement[29]

[29] `http://127.0.0.1:20022/q-ab-recursivite-liste-1.quizz`

11. Itérateurs sur les listes

Les itérateurs permettent d'appliquer un traitement (une fonction) à tous les termes d'une liste. Ils sont construits selon le schéma de récursion simple sur les listes.

11.1. La fonction `filtre`

Étant donnée une liste L d'entiers, supposons que l'on veuille rendre la liste des éléments de L qui sont pairs. Par exemple :

```
(filtre-pairs (list 1 2 3 5 8 6)) → (2 8 6)
```

On peut écrire la fonction suivante :

```
;;; filtre-pairs : LISTE[int] -> LISTE[int]
;;; (filtre-pairs L) rend la liste des éléments de «L» qui sont pairs
(define (filtre-pairs L)
  (if (pair? L)
      (if (even? (car L))
          (cons (car L) (filtre-pairs (cdr L)))
          (filtre-pairs (cdr L)))
      (list)))
```

Si l'on veut rendre la liste des éléments de L qui sont impairs, il faut écrire une autre fonction :

```
;;; filtre-impairs : LISTE[int] -> LISTE[int]
;;; (filtre-impairs L) rend la liste des éléments de «L» qui sont impairs
(define (filtre-impairs L)
  (if (pair? L)
      (if (odd? (car L))
          (cons (car L) (filtre-impairs (cdr L)))
          (filtre-impairs (cdr L)))
      (list)))
```

En fait, toutes ces définitions ont la même forme, autrement dit elles suivent le même schéma :

```
;;; fonction: LISTE[alpha] -> LISTE[alpha]
(define (fonction L)
  (if (pair? L)
      (if (test ? (car L))
          (cons (car L) (fonction (cdr L)))
          (fonction (cdr L)))
      (list)))
```

et on pourrait dire : « étant donnée une fonction de test (qui a comme donnée un élément de type alpha et qui rend un booléen), une définition de la fonction qui rend la liste des éléménts d'une liste donnée qui vérifient ce test peut être obtenue en écrivant... ».

Mais, plutôt que d'écrire une nouvelle fonction à chaque fois que l'on change le prédicat de filtrage, il vaut mieux écrire une fonction, paramétrée par le prédicat de filtrage, `filtre` (une telle fonction est dite générique) :

```
;;; filtre : (alpha -> bool) * LISTE[alpha] -> LISTE[alpha]
;;; (filtre test? L) rend la liste des éléments de la liste «L»
;;; qui vérifient le prédicat «test?».
(define (filtre test? L)
  (if (pair? L)
      (if (test? (car L))
```

[30] http://127.0.0.1:20022/q-ab-recursivite-liste-2.quizz

```
            (cons (car L)
                  (filtre test? (cdr L)))
          (filtre test? (cdr L)))
      (list)))
```

Noter que le premier argument de cette fonction est une fonction – plus précisément un prédicat – (on dit que cette fonction est une fonctionnelle).

Et alors :

```
(filtre even? (list 1 2 3 4 5)) → (2 4)
(filtre odd? (list 1 2 3 4 5)) → (1 3 5)
(filtre integer? (list 2 2.5 3 3.5 4)) → (2 3 4)
```

Dans tous ces exemples, la fonction de test est une fonction prédéfinie. Comment faire lorsque ce n'est pas le cas ? Par exemple, comment filtrer dans une liste de nombres ceux qui sont supérieurs à un nombre donné ? Il suffit d'écrire un prédicat ad-hoc et appliquer la fonction `filtre` à ce prédicat :

```
;;; sup-4? : Nombre -> Nombre
;;; (sup-4? x) rend vrai ssi «x» est strictement supérieur à 4
(define (sup-4? x)
  (> x 4))
  (filtre sup-4? (list 5 2.5 4 4.5 3)) → (5 4.5)
```

11.2. La fonction `map`

On veut écrire une définition de la fonction qui rend la liste des carrés des éléments d'une liste de nombres. Par exemple :

```
(liste-carres (list 1 2 3 4)) → (1 4 9 16)
```

Après avoir défini la fonction qui rend le carré de sa donnée :

```
;;; carre : Nombre -> Nombre
;;; (carre x) rend le carré de «x»
(define (carre x)
  (* x x))
```

on peut écrire une définition de la fonction `liste-carres` :

```
;;; liste-carres : LISTE[Nombre] -> LISTE[Nombre]
;;; (liste-carres L) rend la liste des carrés des éléments de «L»
(define (liste-carres L)
  (if (pair? L)
      (cons (carre (car L))
            (liste-carres (cdr L)))
      (list)))
```

On peut aussi vouloir écrire une définition de la fonction qui rend la liste des racines carrées des éléments d'une liste de nombres. Par exemple :

```
(liste-racines-carrees (list 16 25 36)) → (4 5 6)
```

voici une définition de la fonction `liste-racines-carrees` :

```
;;; liste-racines-carrees : LISTE[Nombre] -> LISTE[Nombre]
;;; (liste-racines-carrees L) rend la liste des racines carrées des
;;; éléments de «L»
(define (liste-racines-carrees L)
  (if (pair? L)
      (cons (sqrt (car L))
            (liste-racines-carrees (cdr L)))
      (list)))
```

Toutes ces définitions suivent le même schéma : la fonction fn-sur-element ayant comme interface :

```
;;; fn-sur-element: alpha -> beta
```

la définition de la fonction est :

```
;;; fonction: LISTE[alpha] -> LISTE[beta]
(define (fonction L)
  (if (pair? L)
      (cons (fn-sur-element (car L))
            (fonction (cdr L)))
      (list)))
```

On peut encore écrire une fonction générique pour mettre en œuvre toutes ces définitions :

```
;;; map : (alpha -> beta) * LISTE[alpha] -> LISTE[beta]
;;; (map fn L) rend la liste dont les éléments résultent de l'application de
;;; la fonction «fn» aux éléments de «L»
(define (map fn L)
  (if (pair? L)
      (cons (fn (car L))
            (map fn (cdr L)))
      (list)))
```

La fonction `map` reçoit une fonction et une liste, et renvoie une liste de même taille, dans laquelle chaque terme est le résultat de l'application de la fonction.

En utilisant cette fonction, voici la mise en œuvre des exemples précédents :

```
(map carre (list 1 2 3 4)) → (1 4 9 16))
(map sqrt (list 16 25 36)) → (4 5 6)
```

et quelques autres exemples d'utilisation :

```
(map even? (list 1 2 3 4 5)) → (#F #T #F #T #F)
(map odd? (list 1 2 3 4 5)) → (#T #F #T #F #T)
```

Remarque : Noter bien la différence entre `map` et `filtre` (qui, toutes les deux, ont comme données une fonction et une liste, la fonction ayant comme type de données le type des éléments de la liste donnée, et rendent une liste) :

- pour `map` , le résultat de la fonction donnée en argument est d'un type quelconque alors que, pour `filtre` , le résultat de la fonction donnée en argument est obligatoirement un booléen ;
- la liste résultat de `map` est de même longueur que sa liste donnée alors que la liste résultat de `filtre` est d'une longueur inférieure ou égale à celle de sa liste donnée ;
- les éléments de la liste résultat de `filtre` sont des éléments de sa liste donnée alors que les éléments de la liste résultat de `map` sont différents de ceux de sa liste donnée.

Un dernier exemple :

```
(map list (list 1 2 3 4)) → ((1) (2) (3) (4))
```

Remarque

La fonction `map` (mais pas `filtre`) est une primitive de Scheme.

11.3. La fonction `reduce`

Parmi les exemples de récursion simple sur les listes, nous avons vu la fonction `somme` qui somme tous les éléments d'une liste de nombres. Nous pourrions aussi définir la fonction `produit` qui rend le produit de tous éléments d'une liste de nombres...

Un exemple dont l'interface est plus complexe : on a, au départ, une somme de *n* francs, on a des rentrées et des dépenses, et on veut connaître le nouveau solde :

```
(solde 300 (list 100 -60 -30)) → 310
(solde 300 (list 100.1 -60.2 -30.4)) → 309.5
```

Cette fonction peut être définie comme suit :

```
;;;  solde : float * LISTE[float] -> float
;;;  (solde s L) rend la somme de «s» et de la somme des éléments de la liste «L»
(define  (solde  s L)
  (if  (pair?  L)
       (+ (car  L) (solde  s (cdr  L)))
       s))
```

Notons que `(somme L)` est alors égal à `(solde 0 L)` : la fonction `solde` est plus générale que la fonction `somme`.

On fait plus compliqué ? Oui : le compte est un compte bancaire et l'informaticien de service a décidé de mettre les centimes dans sa poche (bien sûr le compte initial est alors un entier) :

```
(solde-b  300 (list  100.1  -60.2  -30.4))  → 308
```

La fonction `solde-b` peut être définie, exactement comme la fonction `solde` mais en utilisant, à la place de la fonction `+`, la fonction `add-b` de spécification :

```
;;;  add-b : float * int -> int
;;;  (add-b x n) rend l'entier inférieur à la somme de «x» et de «n»
```

et la définition de `solde-b` est alors :

```
;;;  solde-b : int * LISTE[float] -> int
;;;  (solde-b s L) rend la somme de «s» et de la pseudo-somme des éléments de
;;;  la liste «L» (toutes les sommes étant effectuées en arrondissant à
;;;  l'entier inférieur)
(define  (solde-b  s L)
  (if  (pair?  L)
       (add-b  (car  L) (solde-b  s (cdr  L)))
       s))
```

Noter bien le type des deux fonctions :

- `add-b` est de type `float * int -> int` ;
- `solde-b` est alors obligatoirement de type `int * LISTE[float] -> int`.

Remarque : `floor` et `inexact->exact` étant des fonctions prédéfinies de Scheme de spécifications :

```
;;;  floor : float -> float
;;;  (floor x) rend la valeur réelle égale à l'entier immédiatement inférieur à «x».
;;;  Par exemple (floor 3.2) rend 3.0 et (floor -3.2) rend -4.0

;;;  inexact->exact : Nombre -> Nombre
;;;  (inexact->exact x) rend la valeur exacte (entière ou rationelle) égale à «x».
;;;  Par exemple,  (inexact->exact 3.0) rend 3
```

la fonction `add-b` peut être définie par :

```
;;;  add-b : float * int -> int
;;;  (add-b x n) rend l'entier inférieur à la somme de «x» et de «n»
(define  (add-b  x n)
  (inexact->exact  (+ (floor  x) n)))
```

Généralisons : la fonction `reduce` reçoit une fonction binaire (de type $\alpha \times \beta \to \beta$), une valeur de départ (de type β) et une liste (de type `LISTE[` α`]`), et condense la liste en un unique résultat (de type β), en composant la fonction binaire sur les termes de la liste. Exemples d'utilisation :

```
(reduce  + 0 (list  1 2 3 4 5))  → 15
(reduce  * 1 (list  1 2 3 4 5))  → 120
(reduce  + 300  (list  100.1  -60.2  -30.4))  → 309.5
(reduce  add-b  300  (list  100.1  -60.2  -30.4))  → 308
```

La fonction `reduce` peut être définie par :

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

```
;;; (reduce fn base L) rend la valeur résultant de la composition de la
;;; fonction binaire «fn» sur les éléments de «L», à partir de l'élément «base».
;;; Par exemple (reduce f e (list e1 e2)) == (f e1 (f e2 e))
(define (reduce fn base L)
  (if (pair? L)
      (fn (car L) (reduce fn base (cdr L)))
      base))
```

On peut donner des exemples compliqués :

```
(reduce cons (list) (list 1 2 3 4)) → (1 2 3 4)
(reduce cons (list 4 5) (list 1 2 3)) → (1 2 3 4 5)
(reduce + 0 (map carre (list 1 2 3 4))) → 30
```

et encore plus compliqués :

```
(reduce et #T (map even? (list 1 2 3 4 5 6 7))) → #F
(reduce et #T (map odd? (list 1 3 5 7))) → #T
```

la fonction `et` étant définie par :

```
;;; et : bool * bool -> bool
;;; (et a b) rend #t ssi «a» et «b» sont égaux à #t
(define (et a b)
  (and a b))
```

Remarque : dans l'application de la fonction `reduce`, on ne peut pas utiliser `and` car c'est une forme spéciale et non une fonction.

Un dernier exemple qui utilise les trois itérateurs que nous avons vus (et qui calcule la somme des carrés des éléments pairs de la liste donnée) :

```
(reduce + 0
        (map carre
             (filtre even? (list 1 2 3 4 5 6)))) → 56
```

Pour s'auto-évaluer
Exercices d'assouplissement[31]

Pour s'auto-évaluer
Exercices d'assouplissement[32]

Pour s'auto-évaluer
Exercices d'assouplissement[33]
Questions de cours[34]
Approfondissement[35]

12. Notion de n-uplet

Un *n-uplet* est une structure de données qui comporte *n* éléments, le premier étant d'un certain type (toujours le même, par exemple `alpha`), le deuxième d'un autre type (toujours le même, par exemple `beta`), le troisième d'un autre type (toujours le même, par exemple `gamma`)... Nous noterons `NUPLET[alpha beta gamma ...]` le type de tels n-uplets.

On peut dire aussi qu'un n-uplet est un *enregistrement*, constitué de *n champs*, chacun étant d'un type bien défini.

Par exemple, on peut définir le type

[31] http://127.0.0.1:20022/q-ab-iterateur-1-1.quizz
[32] http://127.0.0.1:20022/q-ab-iterateur-2-1.quizz
[33] http://127.0.0.1:20022/q-ab-iterateur-3-1.quizz
[34] http://127.0.0.1:20022/q-ab-iterateur-3-2.quizz
[35] http://127.0.0.1:20022/q-ab-iterateur-3-3.quizz

NUPLET[nat string bool]

qui pourrait correspondre à un enregistrement regroupant différents noms équivalents pour une valeur booléenne et qui comporte trois champs,

- le premier – que l'on pourrait nommer *naturel* – étant de type `nat` ,
- le deuxième – que l'on pourrait nommer *chaine* – étant de type `string` ,
- le troisième – que l'on pourrait nommer *booleen* – étant de type `bool` .

Attention : ne pas confondre `LISTE` et `NUPLET` :

- en ce qui concerne l'utilisation :
  - les listes correspondent aux séquences finies,
  - les n-uplets correspondent à des enregistrements ;
- pour le nombre d'éléments :
  - deux listes de type `LISTE[ α]` peuvent avoir des nombres d'éléments différents,
  - deux n-uplets de type `NUPLET[...]` ont exactement le même nombre d'éléments ;
- pour le type des éléments :
  - tous les éléments d'une liste de type `LISTE[ α]` sont de même type (à savoir α),
  - les différents éléments d'un n-uplet de type `NUPLET[...]` peuvent être de types différents (et sont, en général, de types différents).

12.1. Fonctions de base pour les n-uplets

12.1.1. Constructeur

On doit pouvoir « fabriquer » un n-uplet. Par exemple, pour le type `NUPLET[nat string bool]` , on définit la fonction `construction` telle que, par exemple :

`(construction 1 "true" #t)` →`(1 true #t)`

Cette fonction peut être définie en utilisant la primitive `list` :

```
;;; construction : nat * string * bool -> NUPLET[nat string bool]
;;; (construction a b c) rend le triplet «(a b c)»
(define (construction a b c)
  (list a b c))
```

12.1.2. Accesseurs

On doit pouvoir « accéder » à chacun des éléments (ou « extraire » la valeur d'un champ) d'un n-uplet. Ainsi, pour le type `NUPLET[nat string bool]` , on définit trois fonctions qui extraient respectivement le premier, le deuxième et le troisième éléments.

Par exemple, si `noms-vrai` est l'élément égal à `(construction 1 "true" #t)` :

- pour le premier élément :

  `(extraction-naturel noms-vrai)` → `1`

- pour le deuxième élément :

  `(extraction-chaine noms-vrai)` → `"true"`

- pour le troisième élément :

  `(extraction-booleen noms-vrai)` → `#T`

Les définitions de ces fonctions d'extraction s'écrivent facilement à l'aide des fonctions `car` et `cdr` (et des abréviations `cadr` , `caddr` ...) :

- pour le premier élément :

```
;;; extraction-naturel : NUPLET[nat string bool] -> nat
;;; (extraction-naturel t) rend «a» lorsque «t» est le triplet «(a b c)»
(define (extraction-naturel t)
  (car t))
```

- pour le deuxième élément :

```
;;;  extraction-chaine : NUPLET[nat string bool] -> string
;;;  (extraction-chaine t) rend «b» lorsque «t» est le triplet «(a b c)»
(define  (extraction-chaine  t)
  (cadr  t))
```

– et, enfi n, pour le troisième élément :

```
;;;  extraction-booleen : NUPLET[nat string bool] -> bool
;;;  (extraction-booleen t) rend «c» lorsque «t» est le triplet «(a b c)»
(define  (extraction-booleen  t)
  (caddr  t))
```

Pour extraire le $i+1^{ème}$, on utilise la fonction `cad`^{i}`r`, `d`^{i} exprimant que l'on écrit *i* fois la lettre `d`.

Remarque : classiquement,

– la fonction de construction est souvent nommée par le nom du type de l'élément que l'on construit (dans l'exemple précédent, puisqu'un n-uplet correspond à un enregistrement regroupant différents noms équivalents pour une valeur booléenne, la fonction `construction` serait ainsi nommée `noms-valeur-booleenne`),

– les fonctions d'extraction sont souvent nommées par le nom du champ de l'élément que l'on extrait (dans l'exemple précédent, la fonction `extraction-naturel` serait ainsi nommée `naturel`).

Ainsi, les défi nitions des différentes fonctions seraient :

```
;;;;  Noms-valeur-booleenne est le type égal à NUPLET[nat string bool]
;;;  noms-valeur-booleenne : nat * string * bool -> Noms-valeur-booleenne
;;;  (noms-valeur-booleenne a b c) rend le triplet «(a b c)»
(define  (noms-valeur-booleenne  a b c)
  (list  a b c))
;;;  naturel : Noms-valeur-booleenne -> nat
;;;  (naturel t) rend «a» lorsque «t» est le triplet «(a b c)»
(define  (naturel  t)
  (car  t))
;;;  chaine : Noms-valeur-booleenne -> string
;;;  (chaine t) rend «b» lorsque «t» est le triplet «(a b c)»
(define  (chaine  t)
  (cadr  t))
;;;  booleen : Noms-valeur-booleenne -> bool
;;;  (booleen t) rend «c» lorsque «t» est le triplet «(a b c)»
(define  (booleen  t)
  (caddr  t))
```

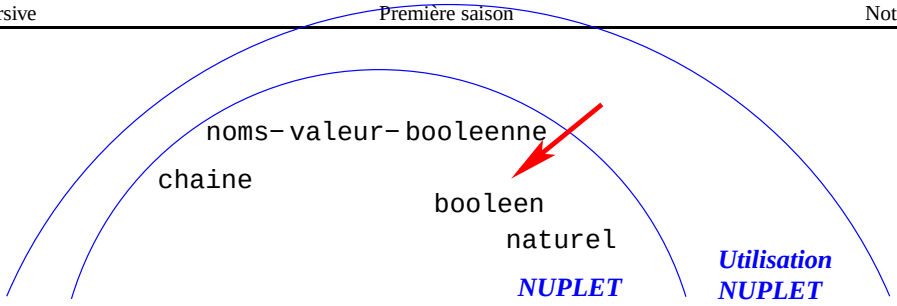
12.2. Notion de niveaux d'abstraction (premier regard)

On peut remarquer que nous avons implanté les types `LISTE`[α] et `NUPLET` [α] en utilisant dans les deux cas les fonctions `list` , `car` et `cdr` . Ceci peut prêter à confusion et c'est pourquoi, dès le début, nous avons insisté sur les différences entre ces deux types.

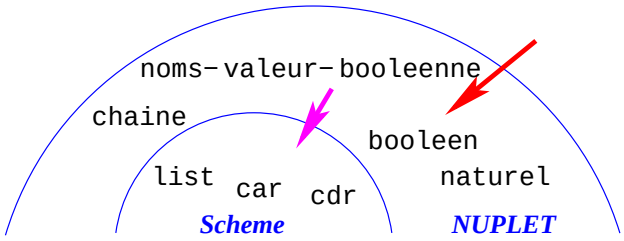
En fait, les notions de `LISTE` et de `NUPLET` n'existent pas dans Scheme. C'est nous qui les défi nissons dans le cadre de ce cours et qui les utilisons. Mais travaillant en Scheme, nous devons implanter ces notions en Scheme et, pour ce faire, nous avons utilisé la même structure de données Scheme, à savoir la « structure de liste » (qui n'est pas équivalente à notre notion de `LISTE`).

Mais nous aurions pu aussi utiliser d'autres structures de données Scheme pour implanter ces types (nous verrons, lors de la troisième saison, une autre implantation des `NUPLET`). Ainsi, lorsque nous parlons de `LISTE` ou de `NUPLET` , nous sommes à un niveau d'abstraction plus élevé.

Plus concrètement, supposons que, dans l'écriture d'un programme, nous ayons besoin de manipuler des noms de valeurs booléennes, triplets de valeurs de types ⟨nat, string, bool⟩ et, pour ce faire, nous avons vu que nous avions besoin du constructeur et des accesseurs `naturel` , `booleen` et `chaine` . Nous pouvons représenter ce besoin par le schéma suivant où la flèche indique que l'« on se sert de » :



Mais ces fonctions n'existent pas en Scheme. Aussi nous les implantons, en utilisant une certaine structure de données Scheme, sans avoir besoin de nous préoccuper de leur utilisation :



En fusionnant les schémas des deux phases, nous obtenons :

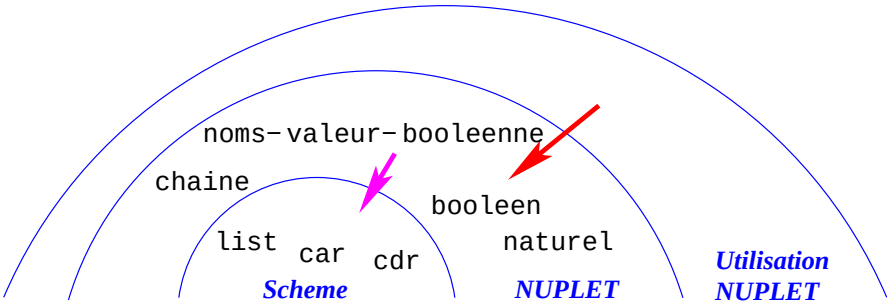


schéma qui montre bien les différents niveaux auquels on peut se placer, niveaux dans lesquels la notion de n-uplet est de plus en plus abstraite lorsque l'on s'éloigne du centre des demi-cercles.
À noter :

1. Nous avons déjà dit que l'on pouvait implanter les n-uplets à l'aide d'autres structures de données Scheme. Aussi – pour faciliter l'évolution du logiciel – est-il recommandé, au niveau de l'utilisation des NUPLET, de ne pas utiliser la connaissance que l'on a de l'implantation retenue. C'est pourquoi, dans notre schéma, il n'y a pas de flèche allant du niveau le plus élevé (« utilisation NUPLET ») au niveau le moins élevé (« Scheme »). Pour insister sur ce point, très important en génie logiciel, on parle de **barrière d'abstraction**. Nous reverrons cette notion de nombreuses fois par la suite.
2. Dans les problèmes – simples – que nous traiterons cette saison, nous ne définirons pas systématiquement les fonctions du niveau d'abstraction « NUPLET » (et nous utiliserons donc les fonctions `car` , `cdr` ... au niveau d'abstraction « utilisation NUPLET »), mais, systématiquement, nous raisonnerons en terme de n-uplets.

Pour s'auto-évaluer
Exercices d'assouplissement[36]
Questions de cours[37]

[36] `http://127.0.0.1:20022/q-ab-nuplet-1.quiz` z
[37] `http://127.0.0.1:20022/q-ab-nuplet-2.quiz` z

13. Notion de semi-prédicat

13.1. Problématique

On peut définir le prédicat `avez-vous?` qui, étant donnés un prédicat et une liste, rend vrai si, et seulement si, il existe une occurrence dans la liste donnée qui vérifie le prédicat donné. Par exemple :

```
(avez-vous?  odd?  (list  4 2 1 3))  → #T
(avez-vous?  odd?  (list  4 2))  → #F
```

Cette fonction se définit très facilement (ce n'est qu'un exercice de révision) :

```
;;;  avez-vous? : (alpha -> bool) * LISTE[alpha] -> bool
;;;  (avez-vous? test? L) rend vrai ssi il existe un élément de la
;;;  liste «L» qui vérifie «test?»
(define  (avez-vous?  test?  L)
  (if  (pair?  L)
       (if  (test?  (car  L))
            #t
            (avez-vous?  test?  (cdr  L)))
       #f))
```

13.2. Semi-prédicat

La fonction précédente fait un peu penser au gag de celui qui répond « oui » lorsqu'on lui demande « avez-vous l'heure ? ». En fait, on peut vouloir définir la fonction `val` qui, étant donnés un prédicat et une liste,

- rend faux lorsqu'il n'existe pas d'élément de la liste donnée qui vérifie le prédicat donné,
- lorsqu'il existe un tel élément, rend une de ses occurrences — choisissons la première — qui vérifient le prédicat donné.

Par exemple :

```
(val  odd?  (list  4 2 1 3))  → 1
(val  odd?  (list  4 2))  → #F
```

Cette fonction, qui rend `#f` dans certains cas et une valeur non booléenne dans d'autres cas, est appellée un **semi-prédicat**. Sa définition peut être :

```
;;;  val : (alpha -> bool) * LISTE[alpha] -> alpha + #f
;;;  (val test? L) rend la valeur du premier élément de «L» qui vérifie «test?»
;;;  s'il existe un tel élément et rend #f sinon.
(define  (val  test?  L)
  (if  (pair?  L)
       (if  (test?  (car  L))
            (car  L)
            (val  test?  (cdr  L)))
       #f))
```

Noter la signature de la fonction avec son « `+ #f` » qui indique un semi-prédicat. Noter aussi que, classiquement, contrairement aux prédicats, les noms des semi-prédicats ne finissent pas par un point d'interrogation.

Un autre exemple : `member` est une primitive de Scheme (cf. carte de référence[38]) dont la spécification est :

```
;;;  member : alpha * LISTE[alpha] -> LISTE[alpha] + #f
;;;  (member v L) rend le suffixe de «L» débutant par la première occurrence de «v»
;;;  ou #f si «v» n'apparaît pas dans «L».
```

Exemples d'application :

```
(member  3 (list  2 3 1 3 2))  → (3 1 3 2)
(member  1 (list  2 3 1))  → (1)
```

[38] http://www.licence.info.upmc.fr/lmd/licence/2005/ue/prec-2005oct/reference.ps

(member 3 (list 2 1)) → #F

La définition pouvant être :

```
(define (member v L)
  (if (pair? L)
      (if (equal? v (car L))
          L
          (member v (cdr L)))
      #f))
```

13.3. Alternative et semi-prédicat

En fait, dans une alternative, toute condition qui n'a pas la valeur `#f` a une valeur de vérité Vrai.

Ainsi, en utilisant la fonction `val` que nous avons définie plus haut :

```
(if (val odd? (list 4 2 1 3))
    "il existe une valeur impaire"
    "il n'existe pas de valeur impaire")
  → il existe une valeur impaire
```

et

```
(if (val odd? (list 4 2))
    "il existe une valeur impaire"
    "il n'existe pas de valeur impaire")
  → il n'existe pas de valeur impaire
```

13.4. Un autre exemple

Nous voudrions définir le semi-prédicat `index` qui, étant donnés un élément et une liste, rend l'index de la première occurrence de cet élément dans la liste s'il existe un tel élément et rend `#f` sinon (dans une liste, l'index de son premier élément est égal à 1, l'index de son deuxième élément est égal à 2...). Par exemple :

```
(index 2 (list 1 2 3)) → 2
(index 4 (list 1 2 3)) → #F
```

Cette fonction peut être définie par :

```
;;; index : alpha * LISTE[alpha] -> nat + #f
;;; (index elm L) rend l'index de la première occurrence de «elm» dans «L»
;;; s'il en existe une (l'index du premier élément d'une liste est égal à 1,
;;; du deuxième élément est égal à 2...) ; rend #f sinon.
(define (index elm L)
  (if (pair? L)
      (if (equal? elm (car L))
          1
          (let ((index-cdr-L (index elm (cdr L))))
            (if index-cdr-L
                (+ 1 index-cdr-L)
                #f)))
      #f))
```

Noter l'utilisation de la variable `index-cdr-L` :

- dans le `if`, elle est considérée comme une condition (qui est vraie lorsque sa valeur n'est pas `#f`) ;
- dans `(+ 1 index-cdr-L)`, on utilise sa « vraie » valeur.

Pour s'auto-évaluer

Exercices d'assouplissement[39]

[39] `http://127.0.0.1:20022/q-ab-semi-predicat -1.quiz z`

14. Liste d'associations

Une liste d'associations est une liste dont chaque terme est une association clef-valeur : chaque terme de la liste d'associations est un couple (c'est-à-dire un n-uplet ayant deux éléments) dont le premier élément est la clef et le second la valeur. Ainsi, étant donnés deux types – `Clef` et `Valeur` – une association est un élément de type `NUPLET[Clef Valeur]` et une liste d'associations est un élément de type `LISTE[NUPLET[Clef Valeur]]` .

Voici un exemple d'association :

```
;;; l'expression suivante est une association de type NUPLET[nat string]
(list 1 "un")
```

(à 1, on associe "un", ou encore la clef est 1 et la valeur est "un").

Un exemple d'une liste d'associations :

```
;;; l'expression suivante est une liste d'associations de type
;;; LISTE[NUPLET[nat string]]
(list (list 1 "un") (list 2 "deux") (list 3 "trois"))
```

(à 1, on associe "un", à 2, on associe "deux", à 3, on associe "trois")

14.1. Ajout dans une liste d'associations

La fonction `ajout` ajoute un couple $\langle cle, valeur \rangle$ à une liste d'associations donnée. Par exemple :

```
(ajout 3 "trois" (list)) → ((3 "trois"))

(ajout 1 "un" (ajout 2 "deux"
                       (ajout 3 "trois" (list))))
  → ((1 "un") (2 "deux") (3 "trois"))

(let * ((L1 (ajout 3 "trois" (list)))
        (L2 (ajout 2 "deux" L1))
        (ma-L (ajout 1 "un" L2)))
  (ajout 2 "two" ma-L))
  → ((2 "two") (1 "un") (2 "deux") (3 "trois"))
```

elle peut être défi nie par :

```
;;; ajout : α * β * LISTE[N-UPLET[α β]] -> LISTE[N-UPLET[α β]]
;;; (ajout clef valeur a-liste) rend la liste d'associations obtenue en ajoutant
;;; l'association « (clef valeur) » en tête de la liste d'associations «a-liste».
(define (ajout clef valeur a-liste)
  (cons (list clef valeur)
        a-liste))
```

14.2. Recherche dans une liste d'associations

La fonction de recherche dans une liste d'associations est une primitive de Scheme, qui a pour nom `assoc` . Son objectif est de rechercher la première association de la liste qui a une clef donnée. Notons que, pour des raisons d'effi cacité, l'ajout d'une nouvelle association se fait en tête de liste. Du coup, lors d'une recherche, on rend toujours l'association la plus récente pour une clef donnée. C'est un semi-prédicat : elle rend #f lorsqu'une telle association n'existe pas, et lorsqu'elle existe, elle rend la valeur « vrai » sous une forme plus informative, à savoir l'association elle-même :

- lorsque la clef n'existe pas :

[40] `http://127.0.0.1:20022/q-ab-semi-predicat -2.quiz z`

```
(assoc  2 (ajout  3 "trois"  (list)))
```
→ #f

- lorsque la clef existe une fois :

```
(let * ((L1  (ajout  3 "trois"  (list)))
        (L2  (ajout  2 "deux"  L1))
        (ma-L  (ajout  1 "un"  L2)))
  (assoc  2 ma-L))
```
→ (2 "deux")

- lorsque la clef existe plusieurs fois :

```
(let * ((L1  (ajout  3 "trois"  (list)))
        (L2  (ajout  2 "deux"  L1))
        (ma-L  (ajout  1 "un"  L2)))
  (assoc  2 (ajout  2 "two"  ma-L)))
```
→ (2 "two")

Rappelons que assoc est une primitive de Scheme. Elle pourrait être défi nie par :

```
;;;  assoc : α * LISTE[N-UPLET[α β]] -> N-UPLET[α β] + #f
;;;  (assoc clef aliste) rend la première association de «aliste» dont le premier
;;;  élément est égal à «clef». Rend la valeur #f en cas d'echec.
(define  (assoc  clef  aliste)
  (if  (pair?  aliste)
       (if  (equal?  clef  (caar  aliste))
            (car  aliste)
            (assoc  clef  (cdr  aliste)))
       #f))
```

On peut aussi vouloir défi nir la fonction valeur-de qui donne la valeur associée à une clef. Par exemple :

```
(valeur-de  2 (ajout  3 "trois"  (list)))  → #F

(let * ((L1  (ajout  3 "trois"  (list)))
        (L2  (ajout  2 "deux"  L1))
        (ma-L  (ajout  1 "un"  L2)))
  (valeur-de  2 ma-L)) → "deux"
```

Cette fonction peut être défi nie par :

```
;;;  valeur-de : α * LISTE[N-UPLET[α β]] -> β + #f
;;;  (valeur-de clef aliste) rend la valeur de la première association de «aliste»
;;;  dont le  premier élément est égal à «clef». Rend #f en cas d'echec.
(define  (valeur-de  clef  aliste)
  (let  ((couple  (assoc  clef  aliste)))
    (if  couple
         (cadr  couple)
         #f)))
```

Rappel : dans une forme conditionnelle, toute condition qui n'a pas la valeur #f a une valeur de vérité Vrai.

14.3. Exemple d'utilisation des listes d'associations

Un dictionnaire français-anglais permet de traduire une phrase française (vue comme une suite de mots) en anglais. Il peut être implanté par une liste d'association :

```
(("chat"  "cat")  ("chien"  "dog")  ("souris"  "mouse"))
```

Le problème est alors de traduire (mot à mot) une phrase française (implantée par une liste de mots) en anglais :

```
(let  ((mon-dico  (list  (list  "chat"  "cat")
                         (list  "chien"  "dog")
```

```
                          (list "souris" "mouse")
                          (list "manger" "eat")
                          (list "fromage" "cheese")))
      (phr1 (list "souris" "manger" "fromage"))
      (phr2 (list "chat" "manger" "souris")))
  (write (traduction mon-dico phr1))
  (write (traduction mon-dico phr2))) → ("mouse" "eat" "cheese")("cat" "eat" "mouse")
```

Avant d'écrire la fonction `traduction`, on peut écrire des fonctions plus simples. Par exemple, `dico-french` rend la liste des mots français du dictionnaire donné :

```
(let ((mon-dico (list (list "chat" "cat")
                      (list "chien" "dog")
                      (list "souris" "mouse"))))
  (dico-french mon-dico)) → ("chat" "chien" "souris")
```

Cette fonction se définit facilement avec un `map` :

```
;;; dico-french : Dico -> LISTE[string]
;;;   où Dico est égal à LISTE[N-UPLET[string string]], le premier string
;;;   étant le mot français et le second string étant le mot anglais
;;; (dico-french dico) rend la liste des mots français du dictionnaire «dico» donné.
(define (dico-french dico)
  (map car dico))
```

Pour définir la fonction `dico-anglais` qui permet de connaître la liste des mots anglais présents dans un dictionnaire, on peut aussi utiliser un `map` :

```
;;; dico-anglais : Dico -> LISTE[string]
;;;   où Dico est égal à LISTE[N-UPLET[string string]], le premier string
;;;   étant le mot français et le second string étant le mot anglais
;;; (dico-anglais dico) rend la liste des mots anglais du dictionnaire «dico» donné.
(define (dico-anglais dico)
  (map cadr mon-dico))
```

Écrivons maintenant la fonction `traduction`. Il suffit de faire un `map` avec la fonction qui traduit un mot. Cette dernière est essentiellement la fonction `valeur-de` que nous avons définie plus haut, le seul problème étant que cette dernière a un argument de trop (le dictionnaire). Pour résoudre ce (petit) problème, il suffit de définir une fonction interne à la définition de la fonction `traduction` :

```
;;; traduction : Dico * LISTE[string] -> LISTE[string]
;;;   où Dico est égal à LISTE[N-UPLET[string string]], le premier string
;;;   étant le mot français et le second string étant le mot anglais
;;; (traduction dico phrase) rend la liste de mots «phrase», traduite selon
;;; le dictionnaire «dico»
(define (traduction dico phrase)
  (define (traduction-mot m)
    (valeur-de m dico))
  (map traduction-mot phrase))
```

Pour s'auto-évaluer

Exercices d'assouplissement[41]
Questions de cours[42]
Approfondissement[43]

[41] http://127.0.0.1:20022/q-ab-liste-association-1. quizz
[42] http://127.0.0.1:20022/q-ab-liste-association-2. quizz
[43] http://127.0.0.1:20022/q-ab-liste-association-3. quizz

15. Citation

15.1. Notions de constante et de symbole

Dans vos programmes Scheme, vous avez utilisé `#f`, `#t`, `12`, `1.2`, `"Scheme est beau"` ... ce sont des **constantes**, les deux premières étant les constantes booléennes, les deux suivantes des constantes numériques et la dernière une constante chaîne de caractères.

<constante> → *<booléen>* `#t` ou `#f`
<nombre> `12`
<chaine> `"DEUG MIAS"`

Mais vous avez aussi utilisé `*`, `ma-L`, `map`, `if` ... ce sont des **symboles**. Noter bien que chaque symbole, même s'il est constitué de plusieurs lettres est considéré comme une entité, Scheme ne regardant pas comment il est « fabriqué ».
Remarques :

1. parmi les symboles précédents, `if` est un mot clef,
2. les autres symboles sont des identificateurs qui identifient des fonctions ou des variables (et on n'a pas le droit d'utiliser un mot clef comme identificateur) ;
3. vous avez aussi utilisé
   – l'espace et le retour chariot qui sont des séparateurs permettant de séparer les différents symboles,
   – les parenthèses,
   – le point-virgule.
4. depuis le début de ce cours, nous avons beaucoup utilisé les chaînes de caractères ; c'est parce que nous n'avions pas la citation à notre disposition ! En fait, presque tous les exemples que nous avons vus, presque tous les exercices que vous avez faits – voire tous –, en « bon » Scheme seraient écrits en utilisant des symboles que l'on citerait.

15.2. Citation

Vous vous êtes peut-être demandé pourquoi, par exemple dans l'exemple donné pour la fonction `append`, nous écrivions les listes données `(list 1 2 3)` et `(list 4 5 6 7)` et que nous disions que la liste résultat était `(1 2 3 4 5 6 7)`. Pourquoi ne peut-on écrire, comme liste donnée `(1 2 3)` ?

En Scheme, programmes et valeurs sont représentés par des listes ou des symboles, la notation `(...)` indiquant une application fonctionnelle ou une forme spéciale. Ainsi, si dans le programme nous écrivons `(1 2 3)`, l'interprète va vouloir appliquer la fonction nommée 1, qui, bien sûr n'existe pas, aux deux arguments 2 et 3. En fait, ce que nous voudrions, c'est **citer** la valeur de la liste `(1 2 3)` à l'intérieur du programme.

Pour pouvoir *citer* des valeurs à l'intérieur d'un programme, on utilise la forme spéciale `quote` ou son abréviation, le caractère apostrophe (`'`) :

<citation> → `(quote` *<donnée>*`)`
`'` *<donnée>*

<donnée> → *<constante>*
<symbole>
`(` *<donnée>** `)`

Attention, `quote` et `'` n'ont pas la même syntaxe : `(quote e)` ≡ `'e`.

Notations : dans ce paragraphe, ≡ sera lu « notation équivalente » et → sera lu « est évalué, par définition du `quote`, en ».

`(quote ab)` ou encore `'ab` désigne le symbole « ab »

`(quote ab)` ≡ `'ab` → `ab`

et `(cons(quote ab)(quote ()))` ou encore `(cons 'ab '())` construit la liste `(ab)`

`(cons(quote ab)(quote ()))` ≡ `(cons 'ab '())` → `(ab)`

La citation d'un nombre est le nombre lui-même :

`(quote 2)` ≡ `'2` → 2

Et il en est de même pour toutes les constantes (booléennes et chaînes de caractères).

La citation d'une liste (suite d'expressions séparées par des espaces et entourées par des parenthèses) est la liste des citations des expressions. Par exemple :

```
   (quote  (a b c)) ≡ '(a b c)
→ (list  'a 'b 'c) → (a b c)

   (quote  (1 2 3)) ≡ '(1 2 3)
→ (list  1 2 3) → (1 2 3)

   (quote  ((1 I) (2 II) (3 III)))
≡ '((1 I) (2 II) (3 III))
→ (list  '(1 I) '(2 II) '(3 III))
→ (list  (list 1 'I) (list 2 'II) (list 3 'III))
→ ((1 I) (2 II) (3 III))
```

Attention : *a priori*, ne pas « quoter » les symboles à l'intérieur d'une liste « quotée ».

15.3. Exemple

Un exemple plus compliqué ? Écrivons une fonction, `calculette`, qui effectue les opérations élémentaires :

```
(calculette  6 '+ 3) → 9
(calculette  6 '* 3) → 18
(calculette  6 '- 3) → 3
(calculette  6 '/ 3) → 2
```

On peut écrire :

```
;;; calculette-0 : Nombre * Operateur * Nombre -> Nombre
;;;   où Operateur est un symbole égal à *, +, - ou /
;;; (calculette-0 arg1 op arg2) rend la valeur
;;; de l'opération désignée par l'opérateur «op» appliquée aux
;;; arguments «arg1» et «arg2»
(define  (calculette-0  arg1 op arg2)
  (cond  ((equal?  op '+) (+ arg1 arg2))
         ((equal?  op '*) (* arg1 arg2))
         ((equal?  op '-) (- arg1 arg2))
         ((equal?  op '/) (/ arg1 arg2))))
```

Remarque : on aurait aussi pu prendre `NUPLET[Nombre Operateur Nombre] -> Nombre` comme profil de la fonction. Excellent exercice !

Mais on peut donner une autre solution. En fait, ce qu'il nous faudrait, c'est une fonction qui associe, à un opérateur (qui est un symbole) l'opération (qui est une fonction) correspondante :

```
;;; operation : Operateur -> Operation
;;;   où Operateur est un symbole égal à *, +, - ou /
;;;   et où Operation est égal à Nombre * Nombre -> Nombre
;;; (operation symbole) rend l'opération associée à l'opérateur donné.
```

et la définition irait alors de soi :

```
;;; calculette : Nombre * Operateur * Nombre -> Nombre
;;;   où Operateur est un symbole égal à *, +, - ou /
;;; (calculette arg1 operateur arg2) rend la valeur
;;; de l'opération désignée par l'opérateur «op» appliquée aux
;;; arguments «arg1» et «arg2»
(define  (calculette  arg1 op arg2)
  ((operation  op)  arg1 arg2))
```

Remarque : noter que `(operation op)` rend une fonction et `((operation op) arg1 arg2)` est une application de cette fonction avec comme arguments `arg1` et `arg2` .

Pour la définition de `operation` , on peut penser à une liste d'associations, les clefs étant les symboles (opérateurs) et les valeurs étant les opérations et l'extraction de l'opération correspondant à un opérateur, est exactement le problème valeur-de que nous avons vu lors de l'étude des listes d'associations :

```
;;; operation : Operateur -> Operation
;;;   où Operateur est un symbole égal à *, +, - ou /
;;;   et où Operation est égal à Nombre * Nombre -> Nombre
;;; (operation symbole) rend l'opération associée à l'opérateur donné.
(define (operation symbole)
  (let ((liste-a (list (list '* *)
                       (list '+ +)
                       (list '- -)
                       (list '/ /))))
    (cadr (assoc symbole liste-a))))
```

Noter l'écriture de la liste d'associations : on ne peut pas écrire cette expression en n'utilisant que le quote (puisque le quote d'une liste quote les éléments de la liste, or dans notre exemple, il y a des éléments des listes qui ne sont pas quotés).

Pour s'auto-évaluer
Exercices d'assouplissement[44]
Questions de cours[45]
Approfondissement[46]

16. Sémantique de Scheme

Jusqu'à présent, nous avons vu les différentes constructions de Scheme en donnant une sémantique (qu'est-ce que fait l'évaluateur ?) intuitive et en regardant (grâce au « pas à pas ») comment l'évaluateur tavaillait. Mais cette façon de faire a ses limites puisque nous ne pouvons pas utiliser le « pas à pas » de DrScheme pour n'importe quel programme (pas de *let*, pas de définition interne…). Aussi, dans cette section, nous voudrions donner une sémantique plus précise et plus générale.

16.1. Idée et problématique

Commençons par une expression simple, lorsque l'on peut utiliser le « pas à pas » :

```
;;; a-disque : Nombre -> Nombre
;;; (a-disque r) rend la surface du disque de rayon «r»
(define (a-disque r)
  (* 3.1416 r r))

;;; essai de a-disque (rend 1520.5344) :
(a-disque (+ 20 2))
```

On peut visualiser le processus de calcul en utilisant DrScheme en mode pas à pas.
Comment çà marche ? On veut calculer :

```
(a-disque (+ 20 2))
```

Dans un premier temps, on doit calculer l'argument de la fonction :

```
(a-disque [(+ 20 2)] )
```

les arguments de la fonction + sont des valeurs et, cette fonction étant une primitive, elle est calculée « d'un coup » :

[44] http://127.0.0.1:20022/q-ab-citation-1.qu izz
[45] http://127.0.0.1:20022/q-ab-citation-2.qu izz
[46] http://127.0.0.1:20022/q-ab-citation-3.qu izz

→ (a-disque 22)

et maintenant que fait-on ?

La fonction `a-disque` n'est pas une primitive, elle a été définie par le programmeur : on prend sa définition, ou plutôt le corps de cette définition, dans laquelle on « remplace » la variable par la valeur de l'argument :

```
(a-disque 22)
→
(* 3.1416 22 22)
```

```
(define (a-disque r)
  (* 3.1416 r r))
```

encore une fois, pour cette application, la fonction est une primitive et les arguments sont des valeurs : le calcul s'effectue immédiatement :

```
(* 3.1416 22 22)
→
1520.5344
```

Terminologie : classiquement, en informatique, on ne dit pas qu'on « remplace » la variable par la valeur de l'argument mais l'on dit que l'on **substitue**, dans le corps de la fonction, la variable par l'argument. La sémantique de Scheme que nous vous présentons est appellée **modèle par substitution**.

16.2. Notion d'environnement

Mais où trouve-t-on la définition de la fonction `a-disque` ? Dans l'environnement dans lequel on évalue l'expression.

En informatique, cette notion d'environnement se décline à tous les niveaux :

- Lorsque l'on travaille sur ordinateur (quoi qu'on fasse), on a besoin d'un certain ensemble de ressources matérielles – par exemple, si l'on veut traiter des images, on a besoin d'un scanner et d'une (bonne) imprimante couleur – et logicielles – par exemple, si vous voulez faire des exercices sur votre ordinateur, vous avez besoin d'un interprète Scheme. C'est l'environnement matériel et logiciel nécessaire.
- Le système d'exploitation qui vous fournit des outils comme la possibilité d'imprimer et qui comporte des **variables d'environnement**, dont la valeur peut varier selon l'utilisateur, selon ce que fait l'utilisateur...
- À l'intérieur de ce système d'exploitation, on peut être dans différents environnements (sous Linux, plusieurs langages de commandes (Shell), plusieurs environnements graphiques...).
- Dans la plupart des logiciels, DrScheme en particulier, on peut être dans différents environnements (niveau d'apprentissage...).
- En Scheme, comme dans tous les langages de programmation, l'évaluation d'une expression est effectuée dans un certain environnement. Ainsi, lorsque l'on définit une fonction, on ajoute cette définition à l'environnement et, dans cet environnement, on peut utiliser cette fonction.

Une autre façon de voir les choses est de dire que l'environnement (quel que soit le niveau où l'on se place) est constitué de choses, de trucs, qui sont indispensables pour effectuer une tâche et que l'on n'a pas besoin de citer explicitement dans la description du traitement à effectuer : il faut d'autant plus avoir conscience de l'existence de cet environnement qu'il est absent du texte descriptif.

16.3. Modèle par substitution

Ainsi, pour évaluer une expression, il faut aussi savoir dans quel environnement on se place.

Au départ – on dit au top-level –, l'environnement est constitué par toutes les primitives (*, +... pair ?, car...). On peut enrichir cet environnement en définissant des fonctions, le nouvel environnement comportant alors les primitives et les fonctions définies par le programmeur.

Ainsi, dans l'exemple précédent, lorsque l'on évalue l'expression `(a-disque (+ 20 2))`, l'environnement est constitué par les primitives et la définition de la fonction `a-disque`.

Premier exemple : on demande l'évaluation de :

```
;;; a-disque : Nombre -> Nombre
;;; (a-disque r) rend la surface du disque de rayon «r»
```

```
(define   (a-disque    r)
  (* 3.14  r r))

;;;  v-cylindre : Nombre * Nombre -> Nombre
;;;  (v-cylindre r h) rend le volume du cylindre de rayon «r»
;;;  et de hauteur «h»
(define   (v-cylindre    r h)
  (* (a-disque    r) h))

;;;  essai de v-cylindre (rend 4559.28) :
(v-cylindre    22 3)
```

En indiquant dans la colonne de gauche l'environnement et dans celle de droite la liste des substitutions, on obtient :

Environnement	*Évaluation*

`(define (a-disque r) (* 3.14 r r))`
`(define (v-cylindre r h) (* (a-disque r) h))`
	`(v-cylindre 22 3)`
	→
	`(* (a-disque 22) 3)`
	→
	`(* (* 3.14 22 22) 3)`
	→
	`(* 1519.76 3)`
	→
	`4559.28`

Second exemple : lorsque nous avons étudié la récursivité sur les listes, nous avons défi ni la fonction « concaténation droite » nommée `conc-d` :

```
;;;  conc-d : LISTE[alpha] * alpha -> LISTE[alpha]
;;;  (conc-d L x) rend la liste obtenue en ajoutant «x» à la fin de la liste «L»
(define   (conc-d   L x)
  (if  (pair?  L)
       (cons  (car  L) (conc-d   (cdr  L) x))
       (list  x)))

(conc-d   '(a  b) 'c)
```

Comment évalue-t-on l'expression (il s'agit d'une révision sur la récursivité) ? L'environnement ne comporte que les primitives et la défi nition de la fonction `conc-d` . L'évaluation est alors :

```
(conc-d   '(a  b) 'c)
```

On substitue, dans la défi nition de `conc-d` , `L` par `'(a b)` et `x` par `'c` :

→

```
(if  (pair?  '(a  b))
     (cons  (car  '(a  b)) (conc-d   (cdr  '(a  b)) 'c))
     (list  'c))
```

L'expression est une alternative. On doit donc évaluer la condition, constituée par une application d'une primitive :

→

```
(if  #t
     (cons  (car  '(a  b))  (conc-d   (cdr  '(a  b))  'c))
     (list  'c))
```

La condition étant vraie, pour évaluer l'alternative, on doit évaluer sa conséquence :

→

```
(cons  (car  '(a  b))  (conc-d  (cdr  '(a  b))  'c))
```

On évalue alors les deux sous-expressions :

→ →

```
(cons  'a  (conc-d  '(b)  'c) )
```

→

Il faut alors évaluer `(conc-d  '(b)  'c)` . Pour ce faire, on substitue, dans la définition de `conc-d` , `L` par `'(b)` et `x` par `'c` :

```
(cons  'a (if  (pair?  '(b))
              (cons  (car  '(b))  (conc-d  (cdr  '(b))  'c))
              (list  'c))
          )
```

L'alternative est évaluée comme précédemment en évaluant la condition :

→

```
(cons  'a (if  #t
              (cons  (car  '(b))  (conc-d  (cdr  '(b))  'c))
              (list  'c))
          )
```

et comme cette dernière est vraie, on évalue la conséquence :

→

```
(cons  'a  (cons  (car  '(b))  (conc-d  (cdr  '(b))  'c)))
```

On évalue les deux expressions `(car  '(b))` et `(cdr  '(b))` :

→ →

```
(cons  'a  (cons  'b  (conc-d  '()  'c)  ))
```

Il faut alors évaluer `(conc-d  '()  'c)` . Pour ce faire, comme d'habitude, on substitue, dans la définition de `conc-d` , `L` par `'()` et `x` par `'c` :

→

```
(cons  'a
       (cons  'b
              (if  (pair?  '())
                   (cons  (car  '())  (conc-d  (cdr  '())  'c))
                   (list  'c))
              ))
```

et on évalue la condition :

→

```
(cons 'a
      (cons 'b
            (if #f
                (cons (car '()) (conc-d (cdr '()) 'c))
                (list 'c))))
```

La condition étant fausse, on évalue l'alternant :

→

```
(cons 'a (cons 'b (list 'c)))
```

et, en deux pas (exécution de la primitive `cons`), on trouve :

→ →

```
(a b c)
```

Pour s'auto-évaluer
Exercices d'assouplissement[47]

17. Définition de fonctions internes – variables globales

17.1. Définition de fonctions internes – variables globales

17.1.1. Définition de fonctions internes

Rappelons les règles de grammaire pour les définitions de fonction :

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

Ainsi, à l'intérieur d'une définition de fonction, on peut définir d'autres fonctions (on parle alors de fonctions internes). Par exemple, on peut donner une autre définition de la fonction `v-cylindre` :

```
;;; v-cylindre : Nombre * Nombre -> Nombre
;;; (v-cylindre r h) rend le volume du cylindre de rayon «r»
;;; et de hauteur «h»
(define (v-cylindre r h)
  ;; a-disque : Nombre -> Nombre
  ;; (a-disque r) rend la surface du disque de rayon «r»
  (define (a-disque r)
    (* 3.14 r r))
  ;; expression de la fonction v-cylindre :
  (* (a-disque r) h))

;;; essai de v-cylindre (rend 4559.28) :
(v-cylindre 22 3)
```

Pour évaluer l'expression donnée, l'environnement ne comporte que la définition de la fonction `v-cylindre` :

Environnement	*Évaluation*

`(define (v-cylindre r h)`
`  (define (a-disque r)`
`  (* 3.14 r r))`
`  (* (a-disque r) h))`
	`(v-cylindre 22 3)`
	→

[47] `http://127.0.0.1:20022/q-ab-semantique-1.` `quizz`

Noter que dans cet environnement, la fonction `a-disque` n'est pas connue : on ne pourrait pas demander à évaluer `(a-disque 22)`. Cela correspond d'ailleurs à l'une des utilisations des définitions internes : toute fonction auxiliaire (*i.e.* que l'on ne définit que pour écrire la définition d'une fonction) doit être une fonction interne.

Pour évaluer l'application de la fonction `v-cylindre`,

- comme précédemment, on subsitue dans le corps de la définition de cette fonction les variables par les arguments correspondants,
- en plus, comme il y a une définition interne, on enrichit l'environnement en lui ajoutant la définition de la fonction `a-disque` :

```
(define (a-disque r)
  (* 3.14 r r))
```

```
(* (a-disque 22) 3)
→
```

le reste de l'évaluation est alors simple :

```
(* (* 3.14 22 22) 3)
→
(* 1519.76 3)
→
4559.28
```

Remarque : à la fin de l'évaluation de l'application de `v-cylindre`, l'environnement retrouve sa valeur initiale et, à nouveau la fonction `a-disque` n'est pas définie.

17.1.2. Autres exemples de fonctions internes

Premier exemple : on demande l'évaluation de :

```
(define (f n)
  (define (g x)
    (+ x x))
  (define (h x)
    (* x x))
  ;; expression de (f n) :
  (+ (g n) (h n) 1))

(f 3)
```

Noter que deux fonctions internes sont définies, au même niveau, dans cet exemple.

Pour évaluer l'expression donnée, l'environnement ne comporte que la définition de la fonction `f` :

Environnement	*Évaluation*

`(define (f n)`
`  (define (g x)`
`  (+ x x))`
`  (define (h x)`
`  (* x x))`
`  (+ (g n) (h n) 1))`
	`(f 3)` →

Pour l'évaluation de l'application `(f 3)`, on substitue 3 à *n* dans la définition de `f` et cette dernière comportant des définitions, on les ajoute à l'environnement :

(define (g x) (+ x x)) (define (h x) (* x x))	(+ (g 3) (h 3) 1) → →

l'évaluation de `(+ (g 3) (h 3) 1)` ne pose alors pas de problème :

(+ (+ 3 3) (* 3 3) 1) → →
(+ 6 9 1) →
16

Deuxième exemple : on demande l'évaluation de :

```
(define (f n)
  (define (g x)
    (define (h y)
      (+ y 2))
    ; expression de (g n) :
    (* x (h x)))
  ;; expression de (f n) :
  (+ (g n) 1))

(f 3)
```

Noter que dans cet exemple, une fonction interne (`h`) est définie à l'intérieur d'une fonction interne (`g`).
Pour évaluer l'expression donnée, l'environnement ne comporte que la définition de la fonction `f` :

Environnement	*Évaluation*
(define (f n) (define (g x) (define (h y) (+ y 2)) (* x (h x))) (+ (g n) 1))	(f 3) →

Pour l'évaluation de l'application `(f 3)`, on substitue 3 à *n* dans la définition de `f` et cette dernière comportant une définition (celle de `g`), on l'ajoute à l'environnement :

(define (g x) (define (h y) (+ y 2)) (* x (h x)))	(+ (g 3) 1) →

Pour l'évaluation de l'application `(g 3)`, on substitue 3 à *x* dans la définition de `g` et cette dernière comportant une définition (celle de `h`), on l'ajoute à l'environnement :

(define (h y) (+ y 2))	(+ (* 3 (h 3)) 1) →

et on substitue alors 3 à *y* dans la définition de `h` :

| `(+ (* 3 (+ 3 2) ) 1)` →...

la fin de l'évaluation est facile et le résultat est :

| →... 16

Troisième exemple : on demande l'évaluation de :

```
(define  (f n)
  (define  (g x)
    (* x (h x)))
  (define  (h y)
    (+ y 2))
  ;; expression de (f n) :
  (+ (h n) (g n)))

(f 3)
```

Noter que dans cet exemple, deux fonctions internes (`g` et `h`) sont définies à l'intérieur de la fonction `f`, l'une des fonctions internes (`h`) étant utilisée dans le corps de l'autre fonction interne (`g`).

Pour évaluer l'expression donnée, l'environnement ne comporte que la définition de la fonction `f` :

Environnement	*Évaluation*

`(define (f n)`
`  (define (g x)`
`  (* x (h x)))`
`  (define (h y)`
`  (+ y 2))`
`  (+ (h n) (g n)))`
	`(f 3)` →

Pour l'évaluation de l'application `(f 3)`, on substitue 3 à *n* dans la définition de `f` et cette dernière comportant deux définitions (celles de `g` et de `h`), on les ajoute à l'environnement :

`(define (g x)`
`  (* x (h x)))`
`  (define (h y)`
`  (+ y 2))`
	`(+ (h 3) (g 3) )` → →

– pour l'évaluation de l'application `(h 3)`, on substitue 3 à *x* dans la définition de `h` ;
– pour l'évaluation de l'application `(g 3)`, on substitue 3 à *y* dans la définition de `g` :

| `(+ (+ 3 2) (* 3 (h 3) ) )` → → |

– l'évaluation de `(+ 3 2)` est immédiate ;
– pour l'évaluation de l'application `(h 3)`, on substitue 3 à *y* dans la définition de `h` :

| `(+ 5 (* 3 (+ 3 2) ))` → |

la fin de l'évaluation est facile et le résultat est :

| →... 20

Quatrième exemple : on demande l'évaluation de :

```
(define  (f n)
  (define  (g x)
    (* x x))
  ;; expression de (f n) :
  (+ (g n) 1))

(/ (f 3) 5)
```

Noter que dans cet exemple, l'application de la fonction `f`, dont la définition comporte une définition interne, est une sous-expression de l'application que l'on doit évaluer.

Pour évaluer l'expression donnée, l'environnement ne comporte que la définition de la fonction `f` :

Environnement	*Évaluation*

`(define (f n)`
`  (define (g x)`
`  (* x x))`
`  (+ (g n) 1))`
	`(/ [(f 3)] 5)` →

Pour l'évaluation de l'application `(f 3)`, on substitue 3 à *n* dans la définition de `f` et cette dernière comportant une définition (celle de `g`), on l'ajoute à l'environnement :

`(define (g x)` `  (* x x))`	⊕	`(/ (+ [(g 3)] 1) 5)` →

Pour l'évaluation de l'application `(g 3)`, on substitue 3 à *x* dans la définition de `g` :

`(/ (+ [(* 3 3)] 1) 5)` →

l'évaluation `(* 3 3)` est immédiate :

`(/ [(+ 9 1)] 5)` →

l'évaluation `(+ 9 1)` termine l'évaluation de `(f 3)`, aussi après cette évaluation, la définition de la fonction `g` n'est plus dans l'environnement :

⊖	`[(/ 10 5)]` →

et le résulat est 2 :

| 2

12.1.1. Notion de variable globale

Dans le paragraphe précédent, nous avons décrit des évaluations d'applications de fonctions dont la définition comporte des définitions internes. Ces définitions sont donc de la forme :

```
(define (f ...)
  (define (g ...)
    ...)
  ;; expression de f :
  ...)
```

Dans ces exemples, hormis les variations d'environnement, les étapes de l'évaluation sont exactement les memes que lorsque les définitions des fonctions `f` et `g` sont au même niveau. Ce n'est pas toujours le cas, comme nous allons voir maintenant.

Premier exemple : donnons une autre définition de la fonction `v-cylindre` :

```
;;; v-cylindre : Nombre * Nombre -> Nombre
;;; (v-cylindre r h) rend le volume du cylindre de rayon «r»
;;; et de hauteur «h»
(define (v-cylindre r h)
  ;; a-disque : Nombre -> Nombre
  ;; (a-disque r) rend la surface du disque de rayon «r»
  (define (a-disque)
    (* 3.14 r r))
  ;; expression de la fonction v-cylindre :
  (* (a-disque) h))

;;; essai de v-cylindre (rend 4559.28) :
(v-cylindre 22 3)
```

Noter que la fonction `a-disque` est une fonction sans argument (vous avez déjà utilisé de telles fonctions, par exemple `(newline)` ou `(list)`).

Noter aussi que dans la définition de la fonction `a-disque` , il y a la variable `r`, qui n'est pas une variable de la fonction, mais une variable de la fonction `v-cylindre` : on dit que `r` est une **variable globale** (on dit aussi variable libre) dans la définition de `a-disque` (et c'est une **variable locale** (on dit aussi variable liée) pour la définition de `v-cylindre`). Remarquer enfin que si l'on « sortait » la définition de la fonction `a-disque` de la définition de `v-cylindre` , il y aurait une erreur car l'évaluateur ne connaîtrait plus `r`, la variable globale (mais locale à la définition de `v-cylindre`).

Comment évalue-t-on l'expression ? Cela commence comme dans l'exemple précédent :

Environnement	*Évaluation*

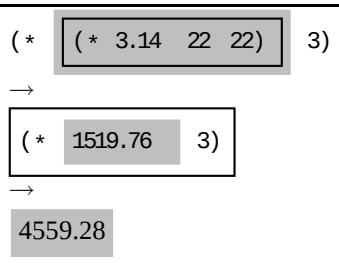
`(define (v-cylindre r h)`
`  (define (a-disque)`
`  (* 3.14 r r))`
`  (* (a-disque) h))`
	`(v-cylindre 22 3)`
	→

et on continue en effectuant de le même façon :

- on subsitue dans le corps de la définition de cette fonction les variables, qui ne sont pas locales à des définitions internes, par les arguments correspondants,
- en plus, comme il y a une définition interne, on enrichit l'environnement en lui ajoutant la définition de la fonction `a-disque` dans laquelle on a substitué 22 à `r` puisque `r` est une variable globale dans la définition de `a-disque` :

`(define (a-disque)` `  (* 3.14 22 22))`	`(* (a-disque) 3)`

le reste de l'évaluation est alors simple :



Second exemple : reprenons la fonction `conc-d` et rappelons la définition que nous avons donnée :

```
(define (conc-d L x)
  (if (pair? L)
      (cons (car L) (conc-d (cdr L) x))
      (list x)))
```

Dans la définition précédente, pour tous les appels récursifs, l'argument correspondant à `x` est toujours le même, celui de l'application initiale de la fonction. Dans cette situation, on peut « globaliser » la variable :

```
;;; conc-d : LISTE[alpha] * alpha -> LISTE[alpha]
;;; (conc-d L x) rend la liste obtenue en ajoutant «x» à la fin de la liste «L»
(define (conc-d L x)
  ;; conc-d-x : LISTE[alpha] -> LISTE[alpha]
  ;; (conc-d-x L) rend la liste obtenue en ajoutant «x» à la fin de la liste «L»
  (define (conc-d-x L)
    (if (pair? L)
        (cons (car L) (conc-d-x (cdr L)))
        (list x)))
  (conc-d-x L))
```

Dans cette définition, `x` est une variable globale dans la définition de `conc-d-x` .

Lors de l'évaluation d'une application de `conc-d` , on commence par enrichir l'environnement en ajoutant la fonction `conc-d-x` pour la valeur de l'argument correspondant à `x` et, dans tous les appels récursifs, on n'a pas besoin de transmettre cet argument, il est directement dans la fonction. Par exemple, pour évaluer l'application

```
(conc-d '(a b) 'c)
```

on commence par enrichir l'environnement avec la fonction `conc-d-x` :

```
(define (conc-d-x L)
  (if (pair? L)
      (cons (car L) (conc-d-x (cdr L)))
      (list 'c)))
```

(noter le `'c` à la place du `x`) et, dans ce nouvel environnement, on évalue :

```
(conc-d-x '(a b))
```

L'évaluation est alors exactement la même que précédemment sauf que toutes les fois où l'on avait `(conc-d ... 'c)` , on a `(conc-d-x ...)` . Autrement dit, on n'a pas besoin de transmettre cet argument à chaque appel récursif car il est mis, une fois pour toutes, dans la définition même de la fonction récursive.

17.1.4. Une autre utilisation des fonctions internes

L'utilisation précédente des fonctions internes – globalisation de variables – n'est pas indispensable (mais permet un gain en terme de nombre de substitutions et donc en efficacité). Dans l'exemple du présent paragraphe, l'utilisation d'une fonction interne, avec variable globale, est indispensable.

Nous voudrions écrire une définition de la fonction qui, étant donnée `L`, une liste non vide de nombres, rend la liste obtenue en multipliant tous les éléments de `(cdr L)` par `(car L)` . Par exemple,

`(mult '(3 4 5 6))` → `(12 15 18)` .

Comment faire ? Si nous devions écrire une fonction qui rend une liste obtenue en multipliant tous les éléments de la liste donnée par une constante, nous utiliserions la fonction `map` , appliquée à une fonction *ad hoc* qui rend le produit de sa donnée par la constante. Dans notre problème, on rend la liste obtenue en multipliant tous les éléments de `(cdr L)` par une valeur ; pas de problème, il suffit que la fonction `map` soit appliquée à `(cdr L)` ! En revanche, le fait que le facteur multiplicatif ne soit pas une constante – c'est `(car L)` – est un problème car cette valeur n'est connue qu'à l'intérieur de l'appel de la fonction `mult` : on doit donc écrire la définition de la fonction *ad hoc* – que nous nommons `mult-elem` – soit écrite à l'intérieur de la définition de la fonction `mult` :

```
;;; mult : LISTE[Nombre]/non vide/ -> LISTE[Nombre]
;;; (mult L) rend la liste obtenue en multipliant tous les éléments de « (cdr L) »
;;; par « (car L) »
(define (mult L)
  ;; mult-elem : Nombre -> Nombre
  ;; (mult-elem e) rend « (* (car L) e) »
  (define (mult-elem e)
    (* (car L) e))
  ;; expression de la définition de (mult L) :
  (map mult-elem (cdr L)))
```

Pour s'auto-évaluer
Exercices d'assouplissement[48]

18. Bloc en Scheme (suite et fin)

Tout d'abord, rappelons les règles de grammaire :

<*bloc*> → `(let (` <*liaison*>* `)` <*corps*> `)`
`(let * (` <*liaison*>* `)` <*corps*> `)`
<*liaison*> → `(` <*variable*> <*expression*> `)`
<*corps*> → <*définition*>*<*expression*>

et donnons un « exemple » :

```
(let ((v1 exp1)
      (v2 exp2))
  (define (f1 ...) corps f1)
  (define (f2 ...) corps f2)
  expression)
```

Précision : nous avons déja dit que l'on ne pouvait pas utiliser les variables liées par les liaisons à l'intérieur des expressions présentes dans les liaisons (par exemple, on ne peut pas utiliser `v1` dans exp_2). En revanche, on peut utiliser les fonctions définies dans toute la partie des définitions (par exemple, on peut utiliser f_1 et f_2 dans *corps* f_1 et dans *corps* f_2 pour effectuer une **récursivité croisée**).

18.1. Sémantique

Pour évaluer un bloc – qui est donc constitué de liaisons, de définitions et d'une expression – dans un certain environnement que nous nommerons, dans les explications suivantes, environnement courant :

1. on évalue, dans l'environnement courant, chacune des expressions présentes dans les liaisons,
2. on substitue, dans les définitions et l'expression, chaque variable liée (*i.e.* présente dans les liaisons) par la valeur de l'expression de sa liason,
3. on enrichit l'environnement courant avec les différentes définitions,
4. et on évalue, dans ce dernier environnement, l'expression.

[48] `http://127.0.0.1:20022/q-ab-var-globale-1.quizz`

Un exemple (uniquement didactique) :

```
(let  ((v1  (+ 1 1))  (v2  (+ 1 1 1)))
  (define  (f1  n)  (+ v1  (f2  n)))
  (define  (f2  n)  (* v2  n))
  (f1  (+ v1  v2)))
```

dans un premier temps, on évalue `(+ 1 1)` et `(+ 1 1 1)` dans l'environnement courant puis on substitue `v1` par 2 et `v2` par 3 dans les définitions et l'expression. Le bloc est ainsi transformé en :

```
(define  (f1  n)  (+ 2  (f2  n)))
(define  (f2  n)  (* 3  n))
(f1  (+ 2  3)))
```

les deux définitions enrichissant l'environnement, et on peut alors évaluer, dans ce dernier environnement, l'expression :

```
(f1 (+ 2 3)) → (f1 5)
             → (+ 2 (f2 5))
             → (+ 2 (* 3 5))
             → (+ 2 15)
             → 17
```

18.2. Utilisation

Ainsi un bloc (qui est une expression) est constitué de trois parties

1. des liaisons variables — valeurs,
2. des définitions de fonctions,
3. une expression.

et toutes les fois où l'on veut

- nommer des valeurs,
- définir des fonctions à l'intérieur d'une expression,

on doit utiliser un bloc.

Exemples

Nous avons déjà vu un grand nombre d'exemples où l'on nomme des valeurs (et nous en verrons deux autres dans le paragraphe suivant), aussi nous ne donnerons que deux exemples, le premier où l'on nomme des valeurs et l'on définit une fonction, le second où l'on ne fait que définir une fonction.

Comme premier exemple, considérons la fonction `mult` dont nous avons déjà donné une définition (avec `map`) et écrivons une autre définition, sans utiliser `map`. Rappelons que, étant donnée une liste de nombres non vide `L`, cette fonction `mult` rend la liste obtenue en multipliant tous les éléments de `(cdr L)` par `(car L)`. Par exemple,

`(mult '(3 4 5 6))` → `(12 15 18)`.

pour écrire une définition de cette fonction, on peut penser

- écrire une fonction interne qui rend la liste voulue,
- pour ce faire, nommer le car de la liste donnée :

```
;;; mult : LISTE[Nombre]/non vide/ -> LISTE[Nombre]
;;; (mult L) rend la liste obtenue en multipliant tous les éléments de « (cdr L) »
;;; par « (car L) »
(define  (mult  L)
  (let  ((facteur  (car  L)))
    ;; mult-aux : LISTE[Nombre] -> LISTE[Nombre]
    ;; (mult-aux L) rend la liste obtenue en multipliant tous les
    ;; éléments de «L» par «facteur»
    (define  (mult-aux  L)
```

```
        (if (pair? L)
            (cons (* facteur (car L))
                  (mult-aux (cdr L)))
            '()))
    (mult-aux (cdr L))))
```

Comme second exemple (définition d'une fonction à l'intérieur d'une expression), considérons la fonction `mult-bis` ayant comme spécification :

```
;;; mult-bis : LISTE[Nombre] -> LISTE[Nombre]
;;; (mult-bis L) rend la liste obtenue en multipliant tous les éléments de «L»
;;; par « (car L) ». Exemples d'applications :
;;;   (mult-bis '(2 3 5)) → (4 6 10)
;;;   (mult-bis '()) → ()
```

Essayons d'écrire une définition de cette fonction en utilisant l'itérateur `map` . Comme nous l'avons déjà vu lorsque nous avons implanté la fonction `map` , pour ce faire, il faut définir une fonction interne qui multiplie sa donnée par `(car L)` . Mais nous ne pouvons pas définir cette fonction directement dans la définition de `mult-bis` car on ne peut le faire que lorsque la liste n'est pas vide. Pour ce faire, il suffit que la conséquence de l'alternative soit un bloc (et comme nous n'avons pas besoin de nommer de valeur, la partie des liaisons est vide) :

```
(define (mult-bis L)
  (if (pair? L)
      (let ()
        (define (mult-elem e)
          (* (car L) e))
        (map mult-elem L))
      '()))
```

18.3. Bloc et efficacité des programmes

Comme nous l'avons vu, la définition de cette dernière fonction peut très bien être écrite sans bloc (en utilisant `map`). Nous allons maintenant étudier des fonctions où l'utilisation d'un bloc est très naturelle et où, surtout, elle est gage d'efficacité.

Tout d'abord, reprenons l'exemple du calcul du nombre de racines d'une équation du second degré :

```
;;; nombre-racines : Nombre * Nombre * Nombre -> nat
;;; (nombre-racines a b c) rend le nombre de racines de l'équation
;;; « a.x**2 + b.x + c = 0 »
(define (nombre-racines a b c)
  (let ((delta (- (* b b) (* 4 a c))))
    (if (< delta 0)
        0
        (if (= delta 0)
            1
            2))))
```

Dans la section 6, nous avons donné cet exemple en insistant sur le fait que c'était très naturel car nous avons l'habitude d'agir ainsi. Aujourd'hui nous voudrions insister sur l'aspect efficacité : l'utilisation du `let` permet de ne calculer qu'une fois (au lieu de 2) l'expression `(- (* b b) (* 4 a c))` .

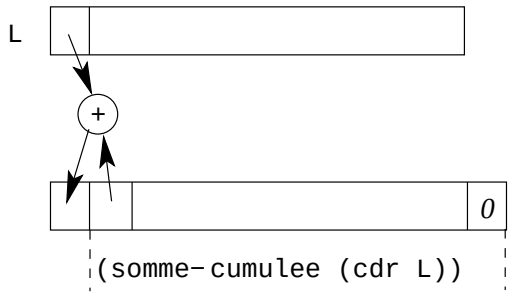
Bien sûr, dans cet exemple, le gain n'est pas très important, mais nous allons voir maintenant deux autres exemples où le fait de ne pas nommer une valeur qu'on utilise plusieurs fois peut être catastrophique pour l'efficacité.

Premier exemple : nous voudrions tout d'abord écrire une définition de la fonction qui a la spécification suivante :

```
;;; somme-cumulee-1 : LISTE[Nombre] -> LISTE[Nombre]
;;; (somme-cumulee-1 L) rend la liste dont le premier élément est égal
;;; à la somme des éléments de «L», dont le deuxième élément est égal à
;;; la somme des éléments de « (cdr L) »... dont l'avant dernier élément est égal
```

;;; *du dernier élément de «L» et dont le dernier élément est nul.*
;;; *Exemple : (somme-cumulee-1 '(1 2 3 4)) → (10 9 7 4 0)*

Idée :



Soit `L` une liste donnée (dans le dessin ci-dessus, elle est schématisée par le rectangle supérieur) et `L-res` la liste $somme-cumulee-1(L)$ (dans le dessin ci-dessus, elle est schématisée par le rectangle inférieur). Alors
- `(cdr L-res)` est égal à `(somme-cumulee-1 (cdr L))` ;
- `(car L-res)` est égal à `(+ (car L) (cadr L-res))` soit à `(+ (car L) (car (somme-cumulee-1 (cdr L))))` .

Solution inefficace :

Si l'on écrit comme corps de la fonction `somme-cumulee-1` l'expression suivante :

```
(if (pair? L)
    (cons (+ (car L) (car (somme-cumulee-1 (cdr L))))
          (somme-cumulee-1 (cdr L)))
    '(0))
```

pour calculer `(somme-cumulee-1 L)`, il faut calculer deux fois `(somme-cumulee-1 (cdr L))`. Bonjour l'efficacité ! En effet, si l'on nomme t_n le temps nécessaire à l'évaluation d'une application de cette fonction sur une liste de longueur n, $t_n = 2 \times t_{n-1} + c$: t_n est de l'ordre de 2^n alors que, visiblement, on peut résoudre ce problème en un temps linéaire comme nous allons le voir maintenant.

Solution efficace :

Il faut nommer la valeur `(somme-cumulee-1 (cdr L))`. Noter que le bloc ne peut pas être le corps de la définition de la fonction car, pour calculer la valeur, on a besoin de `(cdr L)` et donc de savoir que la liste n'est pas vide.

```
(define (somme-cumulee-1 L)
  (if (pair? L)
      (let ((sc-cdrL (somme-cumulee-1 (cdr L))))
        (cons (+ (car L) (car sc-cdrL))
              sc-cdrL))
      '(0)))
```

Pour des raisons d'efficacité, dans une définition récursive, il ne faut jamais appeler plusieurs fois la fonction récursive avec les mêmes arguments. Si l'idée entraîne une telle définition, on doit utiliser un bloc qui lie une variable à la valeur de l'appel. Bien sûr, cela ne veut pas dire qu'il ne faut jamais écrire des définitions ayant plusieurs appels récursifs, mais ceux-ci doivent avoir des arguments différents.

Second exemple : dans l'exemple précédent, nous avons ajouté 0 à la fin de la liste résultat parce que cela facilite l'écriture de la définition. Essayons maintenant d'écrire une définition de la « vraie » fonction somme cumulée :

;;; *somme-cumulee : LISTE[Nombre] -> LISTE[Nombre]*
;;; *(somme-cumulee L) rend la liste dont le premier élément est égal à la*
;;; *somme des éléments de «L», dont le deuxième élément est égal à la somme*
;;; *des éléments de « (cdr L) »... dont le dernier élément est égal au dernier*
;;; *élément de «L». Exemple : (somme-cumulee '(1 2 3 4)) → (10 9 7 4)*

Pour écrire la définition, le problème est alors que `(somme-cumulee (cdr L))` peut être la liste vide (ce qui se passe lorsque `(cdr L)` est la liste vide). On doit donc exclure ce cas de l'appel récursif :

```
(define (somme-cumulee0 L)
  (if (pair? L)
      (if (pair? (cdr L))
          (let ((s-cdrL (somme-cumulee0 (cdr L))))
            (cons (+ (car L) (car s-cdrL))
                  s-cdrL))
          L)
      '()))
```

Mais faisons tourner cette définition à la main. On teste si `L` est la liste vide puis si `(cdr L)` est la liste vide. Dans le cas contraire, on applique récursivement `somme-cumulee0` sur `(cdr L)`, nommons L_1 cette valeur. Pour évaluer cette application, on commencera par tester si L_1 n'est pas vide ; or il ne le sera pas puisque nous avons vu que `(cdr L)` ne l'était pas. Ce test est donc inutile dans les appels récursifs (mais il l'est au départ).

Comment gérer cela ? Il suffit de définir une autre fonction, qui rend la valeur rendue par la fonction que nous définissons, mais que l'on n'appelle que lorsque la liste donnée n'est pas vide. Cette fonction n'est qu'une fonction auxiliaire, aussi nous écrivons sa définition à l'intérieur de la définition demandée :

```
;;; somme-cumulee : LISTE[Nombre] -> LISTE[Nombre]
;;; (somme-cumulee L) rend la liste dont le premier élément est égal à la
;;; somme des éléments de «L», dont le deuxième élément est égal à la somme
;;; des éléments de « (cdr L) »... dont le dernier élément est égal au dernier
;;; élément de «L». Exemple : (somme-cumulee '(1 2 3 4)) → (10 9 7 4)
(define (somme-cumulee L)
  ;; sc-non-vide : LISTE[Nombre]/non vide/ -> LISTE[Nombre]
  ;; (sc-non-vide L) = (somme-cumulee L)
  (define (sc-non-vide L)
    (if (pair? (cdr L))
      (let ((sc-cdrL (sc-non-vide (cdr L))))
        (cons (+ (car L) (car sc-cdrL))
              sc-cdrL))
      L))
  ;; expression de somme-cumulee :
  (if (pair? L)
      (sc-non-vide L)
      '()))
```

Pour s'auto-évaluer
Exercices d'assouplissement[49]
Questions de cours[50]

19. Types string, Ligne et Paragraphe

19.1. String

Nous utilisons des valeurs de type « string » presque depuis le début de ce cours. En effet, en Scheme, lorsque nous écrivons `"toto"`, nous écrivons un « littéral string » (chaîne de caractères en français).

[49] http://127.0.0.1:20022/q-ab-bloc-1.quizz
[50] http://127.0.0.1:20022/q-ab-bloc-2.quizz

19.1.1. Fonctions primitives

Dans la suite du cours, nous écrirons des définitions de fonctions qui manipulent des chaînes de caractères et, pour ce faire, nous utiliserons les primitives Scheme (*i.e.* les fonctions fournies par DrScheme) suivantes :
Pour connaitre la longueur d'une chaîne :

```
;;; string-length : string -> nat
;;; (string-length s) rend la longueur de la chaîne «s»
;;; Par exemple, (string-length "abc") rend 3 et (string-length "") rend 0
```

Une fonction pour concaténer des chaînes :

```
;;; string-append : string * ... -> string
;;; (string-append s1 ...) rend la chaîne de caractères obtenue en mettant
;;; « bout à bout » les différentes chaînes données
;;; Par exemple, (string-append "abc" "de") rend la chaîne "abcde"
```

On peut considérer qu'une chaîne de caractères est une suite de caractères, le premier caractère ayant comme indice 0, le second caractère ayant comme indice 1,... et le dernier caractère ayant comme indice la longueur de la chaîne moins un. On dispose alors d'une fonction qui rend une sous-chaîne de la chaîne donnée, sous-chaîne spécifiée par l'indice de son premier caractère et un de plus que l'indice, dans la chaîne donnée, de son dernier caractère dans la chaîne donnée :

```
;;; substring : string * nat * nat -> string
;;; ERREUR lorsque les indices ne sont pas corrects
;;; (substring s i j) rend la sous-chaîne de caractères de la chaîne «s» d'indices « [i . . . j[ »
;;; Par exemple, (substring "abc" 1 3) rend la chaîne "bc"
```

19.1.2. Exemples

La fonction `prem` rend la chaîne de caractères qui ne comporte qu'un caractère, le premier de la chaîne donnée. Notons que cette spécification n'a de sens que si la chaîne donnée a, au moins, un caractère, c'est-à-dire si elle n'est pas vide :

```
;;; prem : string -> string
;;; ERREUR lorsque la chaîne donnée est la chaîne vide
;;; (prem s) rend la chaîne de caractères qui ne comporte qu'un caractère,
;;; le premier de la chaîne «s» donnée
```

L'implantation est simple, il suffit de considérer la sous-chaîne qui commence au premier caractère (*i.e.* celui d'indice 0) et qui se termine au second caractère (*i.e.* celui d'indice 1) :

```
(define (prem s)
  (substring s 0 1))
```

Considérons maintenant la fonction qui rend la chaîne de caractères obtenue en ôtant, à la chaîne donnée, son premier caractère :

```
;;; sauf-prem : string -> string
;;; ERREUR lorsque la chaîne donnée est la chaîne vide
;;; (sauf-prem s) rend la chaîne de caractères obtenue en ôtant, à la chaîne «s»
;;; donnée, son premier caractère
```

Là encore l'implantation est facile, il suffit de remarquer que l'indice de fin est égal à la longueur de la chaîne :

```
(define (sauf-prem s)
  (substring s 1 (string-length s)))
```

On peut aussi avoir besoin du dernier caractère d'une chaîne donnée :

```
;;; der : string -> string
;;; ERREUR lorsque la chaîne donnée est la chaîne vide
;;; (der s) rend la chaîne de caractères qui ne comporte qu'un caractère,
```

```
;;; le dernier de la chaîne «s» donnée
(define (der s)
  (let ((ls (string-length s)))
    (substring s (- ls 1) ls)))
```

Enfin, considérons la fonction qui rend la chaîne de caractères obtenue en ôtant, à la chaîne donnée, son dernier caractère :

```
;;; sauf-der : string -> string
;;; ERREUR lorsque la chaîne donnée est la chaîne vide
;;; (sauf-der s) rend la chaîne de caractères obtenue en ôtant, à la chaîne «s»
;;; donnée, son dernier caractère


(define (sauf-der s)
  (substring s 0 (- (string-length s) 1)))
```

Comme dernier exemple, considérons la fonction `permutation` :

```
;;; permutation : string -> string
;;; (permutation s) rend la chaîne vide lorsque la chaîne donnée est vide,
;;; sinon rend la chaîne de caractères obtenue en ôtant, à la chaîne «s»
;;; donnée, son premier caractère et en le rajoutant à la fin
;;; Par exemple, (permutation "abc") rend "bca"


(define (permutation s)
  (if (equal? s "")
      ""
      (string-append (sauf-prem s) (prem s))))
```

19.2. Types Ligne et Paragraphe

Les chaînes de caractères vues dans le paragraphe précédent peuvent comporter des caractères « fin de ligne ». Par exemple l'exécution de

```
(permutation "ab
c")
```

rend

```
"b
ca"
```

Par la suite, nous considèrerons le type des chaînes de caractères qui ne comportent pas de caractères « fin de ligne » et nous nommerons « Ligne » le type de ces chaînes. Notons que le type « Ligne » possède donc toutes les fonctions du type « string ».

Un « Paragraphe » est alors une suite de lignes qui seront affichées les unes à la suite des autres, en passant à chaque fois à la ligne. Pour manipuler ces paragraphes, nous utiliserons les fonctions de base suivantes qui ont été ajoutées à la bibliothèque de DrScheme pour cet enseignement et se trouvent mentionnées dans la carte de référence :

```
;;; paragraphe : LISTE[Ligne] -> Paragraphe
;;; (paragraphe L) rend le paragraphe formé des lignes de la liste «L»

;;; paragraphe-cons : Ligne * Paragraphe -> Paragraphe
;;; (paragraphe-cons ligne para) rend le paragraphe dont la première ligne est «ligne»
;;; et dont les lignes suivantes sont constituées par les lignes du paragraphe «para»

;;; lignes : Paragraphe -> LISTE[Ligne]
```

```
;;; (lignes paragraphe) rend la liste des lignes contenues dans le paragraphe donné.
```

19.2.1. Remarque

Les fonctions `paragraphe` et `lignes` vérifient les propriétés suivantes :

Pour tout paragraphe `para` :
`(paragraphe (lignes para))` → `para`

Pour toute liste de lignes `L` :
`(lignes (paragraphe L))` → `L`

19.2.2. Exemple 1

On voudrait écrire la définition d'une fonction qui, étant donnés deux paragraphes, les concatène c'est-à-dire rend le paragraphe qui contient les lignes du premier paragraphe puis les lignes du second paragraphe. Pour implanter cette fonction, on peut

- fabriquer, pour chacun des deux paragraphes, la liste de ses lignes (en utilisant la fonction **lignes**),
- concaténer ces deux listes (en utilisant la fonction **append**),
- fabriquer un paragraphe à partir de cette liste (en utilisant la fonction **paragraphe**) :

```
;;; paragraphe-append : Paragraphe * Paragraphe -> Paragraphe
;;; (paragraphe-append P1 P2) rend le paragraphe qui contient les lignes de «P1»
;;; puis les lignes de «P2»
(define (paragraphe-append P1 P2)
  (paragraphe (append (lignes P1) (lignes P2))))
```

19.2.3. Exemple 2

On voudrait écrire la définition d'une fonction qui, étant donnée une ligne, affiche ses caractères « en triangle » : elle rend un paragraphe dont la première ligne est la ligne donnée, la deuxième ligne est la ligne donnée hormis le dernier caractère, la troisième ligne est la ligne donnée hormis les deux derniers caractères... Par exemple :

```
(triangle1 "abc")
```
rend

```
"
abc
ab
a
"
```

Sa spécification est donc :

```
;;; triangle1 : Ligne -> Paragraphe
;;; (triangle1 ligne) rend un paragraphe dont la première ligne est la ligne
;;; donnée, la deuxième ligne est la ligne donnée hormis le dernier caractère,
;;; la troisième ligne est la ligne donnée hormis les deux derniers caractères...
```

Pour l'implantation de la fonction `triangle1`, remarquons que la première ligne du résultat est la ligne donnée et que les lignes suivantes sont égales à l'application de cette même fonction sur la ligne obtenue en supprimant le dernier caractère de la ligne donnée. On peut donc écrire la définition récursive suivante :

```
(define (triangle1 ligne)
  (if (equal? ligne "")
      (paragraphe '())
      (paragraphe-cons ligne (triangle1 (sauf-der ligne)))))
```

19.2.4. Exemple 3

On voudrait maintenant avoir un triangle « dans l'autre sens » :

(triangle2 "abc")
rend

```
"
abc
 bc
  c
"
```

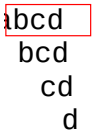
Sa spécifi cation est donc :

```
;;;  triangle2 : Ligne -> Paragraphe
;;;  (triangle2 ligne) rend un paragraphe dont la première ligne est la ligne
;;;  donnée, la deuxième ligne est la ligne donnée hormis le premier caractère,
;;;  la troisième ligne est la ligne donnée hormis les deux premiers caractères...
;;;  toutes ces lignes étant justifiées à droite
```

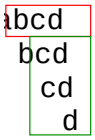
L'implantation est un peu plus délicate. Faisons un dessin : `(triangle2 "abcd")` doit rendre le paragraphe

```
bcd
bcd
 cd
  d
```

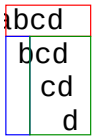
la première ligne de ce paragraphe étant la donnée auquelle on a appliqué la fonction :



et dans les lignes suivantes, on reconnait le résultat de l'application de la fonction à la ligne donnée hormis son premier caractère :



mais en mettant un caractère espace devant chaque ligne :



Ainsi, si l'on dispose d'une fonction `ajout-prefixe` de spécifi cation

```
;;;  ajout-prefixe : Ligne * Paragraphe -> Paragraphe
;;;  (ajout-prefixe pref p) rend le paragraphe obtenu en préfixant chaque ligne
;;;  du paragraphe «p» donné par «pref»
```

la fonction `triangle2` peut être défi nie par :

```
(define (triangle2 ligne)
  (if (equal? ligne "")
      (paragraphe '())
      (paragraphe-cons
       ligne
       (ajout-prefixe " " (triangle2 (sauf-prem ligne))))))
```

Reste à implanter `ajout-prefixe` . Rappelons sa spécification :

```
;;; ajout-prefixe : Ligne * Paragraphe -> Paragraphe
;;; (ajout-prefixe pref p) rend le paragraphe obtenu en préfixant chaque ligne
;;; du paragraphe «p» donné par «pref»
```

Comme il faut concaténer une chaîne devant chaque ligne du paragraphe, il suffit, à partir du paragraphe donné :
- d'extraire la liste de ses lignes (en utilisant la fonction `lignes`),
- de « maper » la fonction qui concatène le préfixe donné à chaque élément de cette liste,
- de reconstituer le paragraphe résultat (en utilisant la fonction `paragraphe`).

D'où la définition :

```
(define (ajout-prefixe pref p)
  ;; ajout-pref : Ligne -> Ligne
  ;; (ajout-pref lig) rend la ligne obtenue en concaténant «pref» devant «lig»
  (define (ajout-pref lig)
    (string-append pref lig))

  ;; expression de (ajout-prefixe pref p)
  (paragraphe (map ajout-pref (lignes p))))
```

Mais, avec la définition précédente, pour calculer `(triangle2 "abcd")` , après avoir calculé `(triangle2 "bcd")` qui renvoie le paragraphe comportant les trois lignes `bcd` , `cd` et `d`, on transforme ce paragraphe en une liste de trois lignes puis on reconstitue un autre paragraphe de trois lignes ; et comme pour calculer `(triangle2 "bcd")` on transforme un paragraphe de deux lignes en une liste de deux lignes et qu'ensuite on reconstitue un paragraphe de deux lignes... Le calcul passe donc son temps à reconstruire un paragraphe à partir d'une liste de lignes obtenue en décomposant un paragraphe.

Pour l'efficacité, on peut donner d'autres définitions qui évitent ces décompositions–recompositions. La définition suivante utilise une fonction auxiliaire qui a une variable de plus que la fonction `triangle2` , variable qui contient une chaîne de caractères qui préfixera toutes les lignes du paragraphe résultat :

```
;;; triangle2 : Ligne -> Paragraphe
;;; (triangle2 ligne) rend un paragraphe dont la première ligne est la ligne
;;; donnée, la deuxième ligne est la ligne donnée hormis le premier caractère,
;;; la troisième ligne est la ligne donnée hormis les deux premiers caractères...
;;; toutes ces lignes étant justifiées à droite
(define (triangle2 ligne)
  ;; triangle2aux : Ligne * Ligne -> Paragraphe
  ;; (triangle2aux pref ligne) rend le paragraphe obtenu en préfixant chaque ligne de
  ;; (triangle2 ligne) par le mot «pref»
  ... à faire
  ... à faire
  ... à faire
  ... à faire
  ... à faire
  ... à faire
  ;; expression de (triangle2 ligne) :
  (triangle2aux "" ligne))
```

Pour l'implantation de la fonction `triangle2aux` , il suffit de remarquer que la première ligne est obtenue en concaténant le préfixe donné et la ligne donnée et que les lignes suivantes sont obtenues en appliquant (récursivement) la fonction
- à un préfixe obtenu en ajoutant un caractère espace au préfixe donné,
- à une ligne obtenue en supprimant le premier caractère de la ligne donnée.

D'où la définition :

```
;;; triangle2 : Ligne -> Paragraphe
;;; (triangle2 ligne) rend un paragraphe dont la première ligne est la ligne
```

```
;;; donnée, la deuxième ligne est la ligne donnée hormis le premier caractère,
;;; la troisième ligne est la ligne donnée hormis les deux premiers caractères...
;;; toutes ces lignes étant justifiées à droite
(define (triangle2 ligne)
  ;; triangle2aux : Ligne * Ligne -> Paragraphe
  ;; (triangle2aux pref ligne) rend le paragraphe obtenu en préfixant chaque ligne de
  ;; (triangle2 ligne) par le mot «pref»
  (define (triangle2aux pref ligne)
    (if (equal? ligne "")
        (paragraphe '())
        (paragraphe-cons (string-append pref ligne)
                         (triangle2aux (string-append " " pref)
                                       (sauf-prem ligne)))))
  ;; expression de (triangle2 ligne) :
  (triangle2aux "" ligne))
```

Pour s'auto-évaluer
Exercices d'assouplissement[51]

[51] http://127.0.0.1:20022/q-ab-fin-saison1-1 .quizz

Seconde saison

Version 1.22

Sommaire

1. Notion de barrière d'abstraction

C'est une notion que vous connaissez déjà !

En effet, pour utiliser une fonction, nous avons dit qu'il fallait regarder sa spécification – et non sa définition –, spécification qui est une description, une « abstraction » de ce que rend la fonction.

```
;;; fn: Donnee -> Resultat
;;; (fn d) rend ...
++++++++++++++++++++++++++
(define (fn d) ...)
```

Ainsi, on peut considérer qu'il y a une barrière entre la fin de la spécification et sa définition proprement dite et, lors d'une utilisation, on n'a pas besoin (et il faudrait même se l'interdire) de franchir cette barrière.

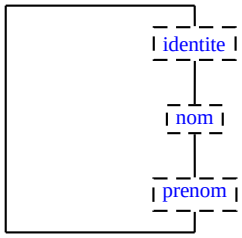
Nous avons vu aussi que, bien entendu, du côté de celui qui écrit la définition de la fonction – on dit qu'il l'implante –, il faut que cette implantation réponde à la spécification.

1.1. Barrière d'abstraction

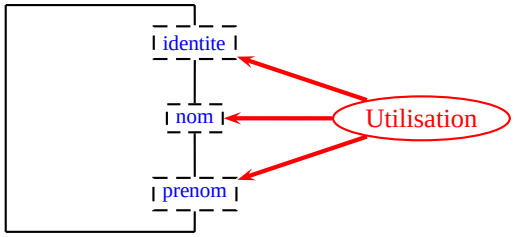
Souvent, on ne peut pas implanter une fonction toute seule sans tenir compte de l'implantation d'autres fonctions, pour avoir un ensemble cohérent.

Aussi, regroupe-t-on toutes ces fonctions dans ce qu'on appelle une **barrière d'abstraction**. Comme pour les fonctions, une barrière d'abstraction a un aspect abstrait (c'est l'ensemble des spécifications des fonctions) et un aspect concret (c'est l'implantation des fonctions).

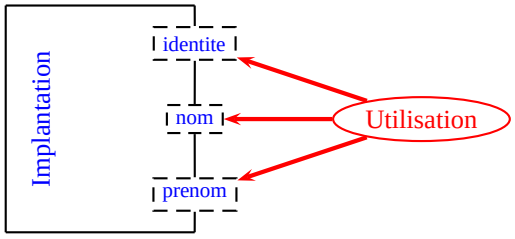
Prenons un exemple (uniquement didactique) : nous voudrions manipuler l'identité des individus, identité qui est composé d'un nom et d'un prénom. Les fonctions utiles sont alors `identite` (qui crée une identité à partir d'un nom et d'un prénom), `prenom` (qui rend le prénom d'une identité donnée) et `nom` (qui rend le nom d'une identité donnée).



Ensuite, lorsque l'on veut définir une fonction dont une donnée ou le résultat est un élément du domaine de la barrière d'abstraction, on n'utilise que les fonctions de la barrière d'abstraction (on s'interdit d'utiliser la connaissance que l'on pourrait avoir de l'implantation de cette barrière d'abstraction) :



Mais, bien entendu, il faut aussi implanter cette abstraction.



L'intérêt (fondamental en génie logiciel) est que l'on peut ensuite changer facilement l'implantation de la barrière d'abstraction (en général pour avoir une implantation plus performante) : il suffit de modifier cette implantation et,

Comme les fonctions utilisatrices de la barrière d'abstraction ne sont définies qu'avec les fonctions de cette barrière, sans utiliser la connaissance des détails de l'implantation, elles restent parfaitement valides.

1.2. Exemple de différentes implantations pour une même barrrière d'abstraction

Prenons l'exemple de l'identité des individus. La spécification de la barrière d'abstraction est :

```
;;;  identite : string * string -> Identite
;;;  (identite nom prenom) rend l'identité dont le nom est «nom» et le
;;;  prénom est «prenom»

;;;  nom : Identite -> string
;;;  (nom id) rend le nom de l'identité «id»

;;;  prenom : Identite -> string
;;;  (prenom id) rend le prénom de l'identité «id»
```

Écrivons une fonction qui, étant donnée une identité, rend la chaîne de caractères composée du prénom de l'individu, suivi d'un espace et terminée par le nom de l'individu :

```
;;;  ident->string : Identite -> string
;;;  (ident->string id) rend la chaîne de caractères composée du prénom de l'individu «id»,
;;;  suivi d'un espace et terminée par le nom de l'individu
(define  (ident->string  id)
  (string-append  (prenom  id)  " "  (nom  id)))
```

Il ne reste plus qu'à implanter la barrière d'abstraction. Vous pensez peut-être qu'il n'y a qu'une solution : un n-uplet qui comporte deux chaînes de caractères. Mais, tout de suite, on voit qu'il y en a deux selon que l'on mette le nom avant ou après le prénom. Ces deux solutions étant très proches, on ne voit pas pourquoi on changerait (mais, lorsque l'on écrit des fonctions sur cette structure de données, il faut savoir dans quel cas on est : il est alors beaucoup plus parlant d'utiliser les fonctions `nom` et `prenom` que les fonctions `car` et `cadr`). Voici cette implantation :

```
;;;  identite : string * string -> Identite
;;;  (identite nom prenom) rend l'identité dont le nom est «nom» et le
;;;  prénom est «prenom»
(define  (identite  nom  prenom)
  (list  nom  prenom))

;;;  nom : Identite -> string
;;;  (nom id) rend le nom de l'identité «id»
(define  (nom  id)
  (car  id))

;;;  prenom : Identite -> string
;;;  (prenom id) rend le prénom de l'identité «id»
(define  (prenom  id)
  (cadr  id))
```

Mais on pourrait aussi utiliser le type `Paragraphe` , en mémorisant le prénom dans la première ligne et le nom dans la seconde ligne :

```
;;;  identite : string * string -> Identite
;;;  (identite nom prenom) rend l'identité dont le nom est «nom» et le
;;;  prénom est «prenom»
(define  (identite  nom  prenom)
  (paragraphe  (list  prenom  nom)))

;;;  nom : Identite -> string
;;;  (nom id) rend le nom de l'identité «id»
(define  (nom  id)
```

```
  (cadr  (lignes  id)))
```

```
;;;  prenom : Identite -> string
;;;  (prenom id) rend le prénom de l'identité «id»
(define  (prenom  id)
  (car  (lignes  id)))
```

Et il y en a (au moins) une quatrième ! L'identité des individus intervient dans tous les fichiers qui comportent des informations sur des personnes, par exemple le fichier de la scolarité qui contient votre cursus, vos notes... Or, dans ce fichier, les identités sont mémorisées sous forme d'une chaîne de caractères obtenue en concaténant le nom, puis une virgule et enfin le prénom. Ainsi, n'est-il pas aberrant d'avoir une autre implantation de cette barrière d'abstraction (nous ne donnons pas les définitions des fonctions `string-avant-virgule` et `string-apres-virgule` car elles sont hors programme) :

```
;;;  identite : string * string -> Identite
;;;  (identite nom prenom) rend l'identité dont le nom est «nom» et le prénom est «prenom»
(define  (identite  nom  prenom)
  (string-append  nom  ","  prenom))

;;;  nom : Identite -> string
;;;  (nom id) rend le nom de l'identité «id»
(define  (nom  id)
  (string-avant-virgule  id))

;;;  prenom : Identite -> string
;;;  (prenom id) rend le prénom de l'identité «id»
(define  (prenom  id)
  (string-apres-virgule  id))
```

avec

```
;;;  string-avant-virgule : string -> string
;;;  (string-avant-virgule s) rend le préfixe de «s» avant la première occurrence du caractère virgule
;;;  HYPOTHÈSE: il y a une occurrrence  du caractère virgule dans «s»

;;;  string-apres-virgule : string -> string
;;;  (string-apres-virgule s) rend le suffixe de «s» après la première occurrence du caractère virgule
;;;  HYPOTHÈSE: il y a une occurrrence  du caractère virgule dans «s»
```

1.3. Mise en œuvre en DEUG MIAS

Par la suite, nous voudrons souvent avoir plusieurs implantations d'une même barrière d'abstraction et tester des fonctions utilisatrices de cette barrière d'abstraction avec chaque implantation de la barrière. Afin de minimiser les « copier–coller », nous utiliserons une architecture logicielle particulière que nous décrivons ci-dessous sur notre exemple.

Ainsi, les définitions Scheme des différentes fonctions de l'exemple ci-dessus sont écrites dans deux fichiers :

- un fichier, nommé `identite1.scm`, contient une première version de la barrière d'abstraction (spécification et implantation),
- un fichier, nommé `identToString.scm`, contient une définition, utilisatrice de cette barrière d"abstraction, de la fonction `ident->string`.

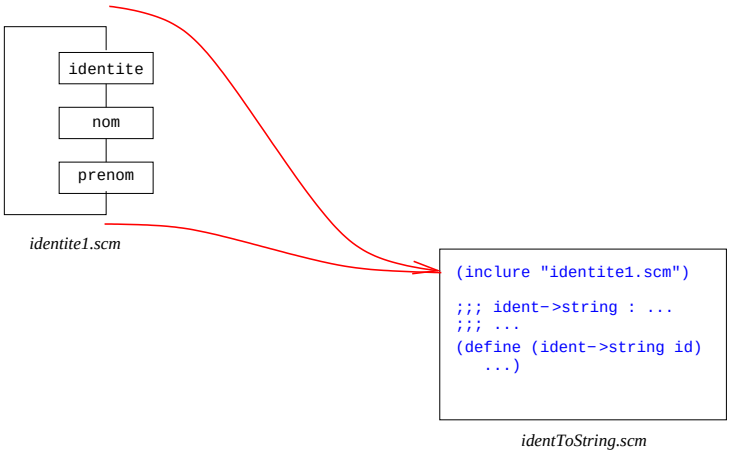
La définition de cette dernière fonction utilisant la barrière d'abstraction, nous demandons que le fichier `identite1.scm` soit inclus en début de fichier grâce à la fonction `inclure` qui a comme spécification :

```
;;;  inclure : string ->
;;;  (inclure s) inclue le fichier de nom «s». Tout se passe comme si le texte
;;;  du fichier de nom «s» était écrit à la place de cette application.
```

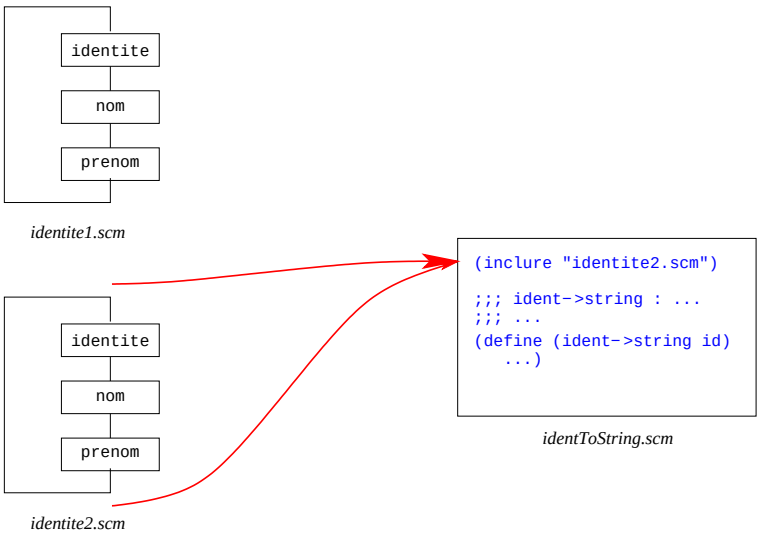
Ainsi, le fichier `identToString.scm` commence par :

```
;;;;  Utilise identite
(inclure  "identite1.scm")
```

On peut schématiser cette architecture par :



Si l'on veut une autre implantation de la barrière d'abstraction, il suffit
- de l'écrire dans un fichier nommé `identite2.scm` et,
- dans le fichier `identToString.scm` de remplacer `(inclure "identite1.scm")` par `(inclure "identite2.scm")` :



Remarque : dans la pratique,
- pour que le test affiche la version de la barrière d'abstraction utilisée, dans le fichier `identToString.scm` ,
  - nous avons défini une fonction qui rend le nom du fichier où se trouve la barrière d'abstraction et
  - nous appliquons cette fonction dans l'application de `inclure` et dans un « display » qui affiche ce nom lors de l'exécution ;
- cette définition est écrite plusieurs fois, avec les noms des fichiers des différentes versions, et nous les commentons systématiquement sauf celle que nous voulons tester.

Ainsi, notre fichier `identToString.scm` (qui permet de tester trois versions de la barrière d'abstraction) est :

```
;;;; Id : identToString.scm,v 1.12002/09/2015 : 18 : 07titouExp
;;;; Utilise identite :
; (define (version-identite) "identite1.scm")
(define (version-identite) "identite2.scm")
```

```
; (define (version-identite) "identite3.scm")
(inclure    (version-identite))

;;;  ident->string : Identite -> string
;;;  (ident->string id) rend la chaîne de caractères composée du prénom de l'individu «id»,
;;;  suivi d'un espace et terminée par le nom de l'individu
(define     (ident->string     id)
  (string-append     (prenom   id)  " "  (nom  id)))

;;;  essai de ident->string :
(display     (string-append     "Essai   avec   "  (version-identite)))
(verifier    ident->string
  (ident->string     (identite    "Curie"    "Pierre"))    ==  "Pierre   Curie")
```

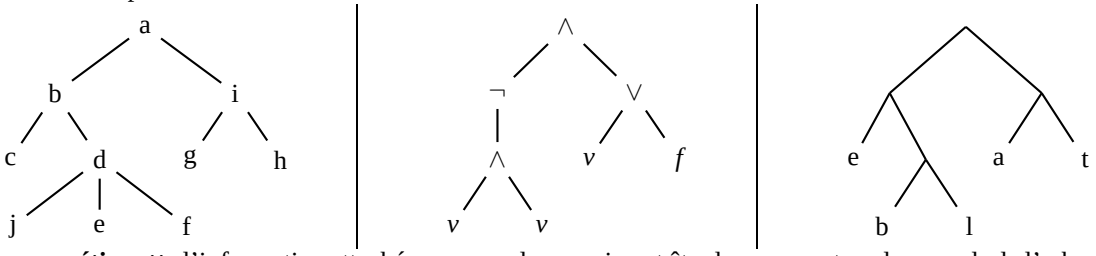
Pour s'auto-évaluer
Exercices d'assouplissement[1]

2. Notion d'arbre

2.1. Notion d'arbre

Un **arbre** est un ensemble de **nœuds** organisés de façon hiérarchique à partir d'un nœud distingué qui est la **racine**, les autres nœuds étant eux-mêmes structurés sous forme d'arbres, les **sous-arbres immédiats** de l'arbre.

Voici trois exemples d'arbres :

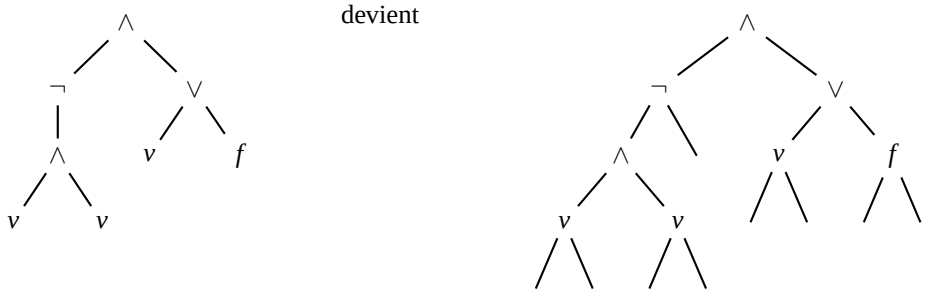


On nomme **étiquette** l'information attachée aux nœuds, ce qui peut être le cas pour tous les nœuds de l'arbre (deux premiers exemples ci-dessous) ou uniquement aux feuilles (dernier exemple ci-dessous).

Intuitivement, les **feuilles** sont les nœuds qui « n'ont rien au-dessous ».

L'arbre dont l'ensemble des nœuds est vide est nommé l'**arbre vide**. Selon l'usage que l'on veut faire des arbres, l'arbre vide peut exister ou non dans l'ensemble des arbres considérés. Ainsi, on peut voir les arbres de deux façons :

- exactement comme le suggère les dessins ci-dessus : il n'y a rien au dessous des feuilles (et une feuille est alors un arbre qui n'a pas de sous-arbre immédiat),
- en considérant que les sous-arbres des feuilles existent et sont (tous) égaux à l'arbre vide. Pour indiquer l'existence de ces arbres vides, dans les dessins, nous ajouterons les traits au-dessous des feuilles :



Un arbre est un **arbre binaire** lorsque c'est l'arbre vide ou lorsqu'il a exactement deux sous-arbres immédiats et que ceux-ci sont eux-mêmes des arbres binaires.

[1] http://127.0.0.1:20022/q-ab-barriere-1.qu izz

2.2. Différentes structures de données « arbre »

La notion d'arbre peut aider pour représenter toute situation hiérarchique :
- sommaire d'un livre (une partie comportant des chapitres qui comportent des sections qui comportent des sous-sections...),
- analyse grammaticale d'une phrase (une phrase est composée d'un sujet, d'un verbe et d'un complément d'objet, le sujet étant un groupe nominale composé d'un article, d'un adjectif et d'un nom, le verbe...),
- classifi cation des animaux,
- arbre généalogique,
- ...

Notons qu'il existe plusieurs « sortes » d'arbres comme structures de données, selon l'endroit où l'on attache les informations (tous les nœuds ou uniquement aux feuilles), selon le nombre de sous-arbres immédiats de chaque nœud (fi xe, borné ou non borné) et selon l'existence ou non de l'arbre vide.

Nous verrons par la suite les « arbres binaires » (information attachée à chaque nœud, existence de l'arbre vide, pour chaque nœud exactement deux sous-arbres immédiats) et les « arbres généraux » (information attachée à chaque nœud, pas d'arbre vide, pour chaque nœud nombre quelconque de sous-arbres immédiats).

3. Barrière d'abstraction des arbres binaires

3.1. Caractéristiques des arbres binaires

Parmi la famille des arbres, les arbres binaires sont caractérisés par :

- information attachée à chaque nœud,
- existence de l'arbre vide,
- chaque nœud a exactement deux sous-arbres immédiats.

3.2. Barrière d'abstraction des arbres binaires

Notation : lorsque le type des étiquettes attachées aux nœuds est α, on notera `ArbreBinaire[ α]` le type des arbres binaires.

Dans ce paragraphe, nous donnons la spécifi cation de la barrière d'abstraction des arbres binaires. Nous vous montrerons une implantation de cette barrière plus tard. Pour que vous puissiez l'utiliser avant de voir cette implantation, nous vous fournissons une autre implantation que vous pouvez utiliser, en TME ou chez vous, sous réserve d'avoir chargé « miastools.plt » (*cf.* installation du céderom).

Dans la bibiothèque MIAS de DrScheme, la barrière d'abstraction des arbres binaires comporte l'ensemble des fonctions (constructeurs, reconnaisseurs et accesseurs) suivantes :

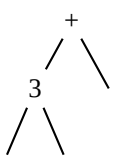
3.2.1. Constructeurs

```
;;; ab-vide : -> ArbreBinaire[α]
;;; (ab-vide) rend l'arbre binaire vide.

;;; ab-noeud : α * ArbreBinaire[α] * ArbreBinaire[α] -> ArbreBinaire[α]
;;; (ab-noeud e B1 B2) rend l'arbre binaire formé de la racine d'étiquette
;;; «e», du sous-arbre gauche «B1» et du sous-arbre droit «B2».
```

Exemples

Pour construire l'arbre suivant :



on peut écrire :

```
(ab-noeud '+
          (ab-noeud 3 (ab-vide) (ab-vide))
          (ab-vide))
```

On ne sait rien de la représentation des arbres ainsi construits. Autrement dit, si vous utilisez l'implantation fournie et si vous tapez une telle expression – toute seule, sans être un argument d'une application –, son évaluation affiche `#<abstract:AB>` qui indique que le résultat est un arbre binaire mais qu'il est manipulé au niveau abstrait, c'est-à-dire que l'on ne veut pas montrer son implantation.

Ainsi, on peut tester les fonctions dont la donnée est un arbre binaire. Pour tester des fonctions dont le résultat est un arbre binaire, la barrière d'abstraction contient une fonction supplémentaire, la fonction `ab-expression`, qui a comme spécification :

```
;;; ab-expression: ArbreBinaire[α] -> Sexpression
;;; (ab-expression B) rend une Sexpression reflétant la construction de l'arbre binaire «B».
```

Autrement dit, cette fonction rend une expression Scheme qui permet de construire (en utilisant les deux constructeurs) l'arbre donné. Par exemple l'application :

```
(ab-expression
 (ab-noeud '+
           (ab-noeud 3 (ab-vide) (ab-vide))
           (ab-vide)))
```

a comme valeur

```
(ab-noeud '+ (ab-noeud 3 (ab-vide) (ab-vide)) (ab-vide))
```

(le résultat est bien l'expression donnée comme argument à la fonction `ab-expression`).

Pour construire des arbres binaires, il est très pratique d'avoir aussi à disposition la fonction `ab-feuille` de spécification :

```
;;; ab-feuille : α -> ArbreBinaire[α]
;;; (ab-feuille e) rend l'arbre binaire constitué d'une feuille, ayant «e» comme étiquette
```

Sa définition peut être :

```
(define (ab-feuille e)
  (ab-noeud e (ab-vide) (ab-vide)))
```

Voici un exemple d'application :

```
(ab-expression
 (ab-noeud '+
           (ab-feuille 3)
           (ab-vide)))
```

a comme valeur

```
(ab-noeud '+ (ab-noeud 3 (ab-vide) (ab-vide)) (ab-vide))
```

La barrière d'abstraction comporte aussi des reconnaisseurs et des accesseurs :

3.2.2. Reconnaisseurs

```
;;; ab-noeud? : ArbreBinaire[α] -> bool
;;; (ab-noeud? B) rend vrai ssi «B» n'est pas l'arbre vide.

;;; ab-vide? : ArbreBinaire[α] -> bool
;;; (ab-vide? B) rend vrai ssi «B» est l'arbre vide.
```

3.2.3. Accesseurs

```
;;; ab-etiquette : ArbreBinaire[α] -> α
;;; (ab-etiquette B) rend l'étiquette de la racine de l'arbre «B»
;;; ERREUR lorsque «B» ne satisfait pas «ab-noeud?»

;;; ab-gauche : ArbreBinaire[α] -> ArbreBinaire[α]
;;; (ab-gauche B) rend le sous-arbre gauche de «B»
;;; ERREUR lorsque «B» ne satisfait pas «ab-noeud?»

;;; ab-droit : ArbreBinaire[α] -> ArbreBinaire[α]
;;; (ab-droit B) rend le sous-arbre droit de «B»
;;; ERREUR lorsque «B» ne satisfait pas «ab-noeud?»
```

3.2.4. Propriétés remarquables

Les fonctions de la barrière d'abstraction des arbres binaires vérifient les propriétés suivantes :

Pour tout couple d'arbres binaires, `G` et `D`, et toute valeur, `v` :

```
(ab-etiquette (ab-noeud v G D)) → v
(ab-gauche (ab-noeud v G D)) → G
(ab-droit (ab-noeud v G D)) → D
```

Pour tout arbre binaire **non vide** `B`

```
(ab-noeud (ab-etiquette B)
          (ab-gauche B)
          (ab-gauche B)) → B
```

3.3. Exemples d'utilisations de la barrière d'abstraction

Pour commencer, on peut définir la fonction `ab-feuille?` dont la spécification est :

```
;;; ab-feuille? : ArbreBinaire[α] -> bool
;;; (ab-feuille? B) rend #t ssi «B» est un arbre binaire réduit à une feuille
;;; ERREUR lorsque l'arbre est vide
```

Sa définition peut être :

```
(define (ab-feuille? B)
  (and (ab-vide? (ab-gauche B))
       (ab-vide? (ab-droit B))))
```

et alors

```
(ab-feuille? (ab-feuille 3))
```

a comme valeur `#t`
et

```
(ab-feuille? (ab-noeud '+
                       (ab-feuille 3)
                       (ab-vide)))
```

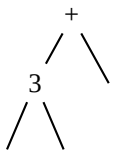
a comme valeur `#f`

3.3.1. Profondeur d'un arbre

On définit récursivement la profondeur d'un arbre binaire comme suit :

- la profondeur de l'arbre vide est 0,
- la profondeur d'un arbre non vide est égale au maximum des profondeurs de ses sous-arbres immédiats, augmenté de 1.

Par exemple, la profondeur de l'arbre



est égale à 2.

Pour écrire une définition de la fonction `ab-profondeur` :

```
;;; ab-profondeur : ArbreBinaire[α] -> nat
;;; (ab-profondeur B) rend la profondeur de l'arbre «B»
```

il suffit de suivre la définition mathématique de la fonction :

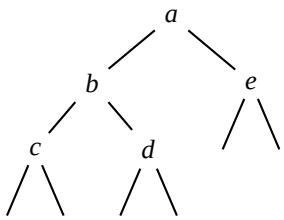
```
(define (ab-profondeur B)
  (if (ab-vide? B)
      0
      (+ 1 (max (ab-profondeur (ab-gauche B))
                (ab-profondeur (ab-droit B))))))
```

3.3.2. Liste infixe des étiquettes d'un arbre

La liste infixe des étiquettes d'un arbre binaire est définie comme suit :

- si `B` est l'arbre vide alors sa liste infixe est vide,
- sinon, la liste infixe de `B` est égale à la concaténation de la liste infixe du sous-arbre gauche de `B` suivi de l'étiquette de la racine de `B` et de la liste infixe du sous-arbre droit de `B`.

Par exemple, la liste infixe de l'arbre



est égale à `(c b d a e)`.

Pour écrire une définition de la fonction

```
;;; ab-liste-infixe : ArbreBinaire[α] -> LISTE[α]
;;; (ab-liste-infixe B) rend la liste infixe des étiquettes de l'arbre «B»
```

il suffit de suivre la définition mathématique de la fonction :

```
(define (ab-liste-infixe B)
  (if (ab-vide? B)
      '()
      (append (ab-liste-infixe (ab-gauche B))
              (cons (ab-etiquette B)
                    (ab-liste-infixe (ab-droit B))))))
```

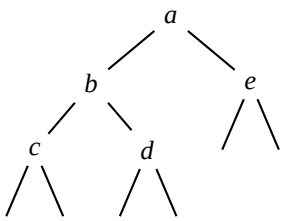
3.3.3. Affichage d'un arbre

Dans ce paragraphe, nous étudions la fonction `ab-affichage` qui permet un affichage plus visuel que la fonction `ab-expression` : `B` étant un arbre binaire, `(ab-affichage B)` rend un paragraphe tel que

- si `B` est l'arbre vide, le paragraphe résultat est composé d'une seule ligne, la ligne `vide` ;
- sinon, le paragraphe résultat est composé

– d'une ligne qui contient l'étiquette de l'arbre,
– les lignes suivantes contiennent la représentation présentement définie du sous-arbre gauche, chaque ligne étant préfixé par le caractère tiret (« - »),
– les lignes suivantes contiennent la représentation présentement définie du sous-arbre droit, chaque ligne étant également préfixé par le caractère tiret (« - »).

Ainsi, si `ex-ab-2` est l'arbre



`(ab-affichage ex-ab-2)` a comme valeur

```
"
a
-b
--c
---vide
---vide
--d
---vide
---vide
-e
--vide
--vide
"
```

Spécification

```
;;; ab-affichage : ArbreBinaire[α] -> Paragraphe
;;; (ab-affichage B) rend, lorsque «B» est l'arbre vide, le paragraphe constitué de la
;;; seule ligne constituée par le mot qui s'affiche "vide" et, lorsque l'arbre «B» n'est pas
;;; vide, le paragraphe dont la première ligne est l'étiquette de la racine de l'arbre «B»
;;; et dont les lignes suivantes sont égales à cette représentation des deux
;;; sous-arbres de «B», toutes les lignes étant préfixées par un tiret
```

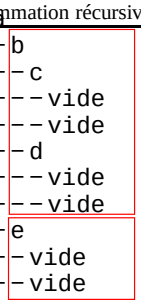
Première implantation

Idée

Considérons l'exemple précédent :

```
a
-b
--c
---vide
---vide
--d
---vide
---vide
-e
--vide
--vide
```

La première ligne est la représentation de l'étiquette de la racine et les lignes suivantes correspondent à la représentation du sous-arbre gauche et du sous-arbre droit, chaque ligne étant précédée d'un tiret :



(ab-affichage (ab-gauche B))

(ab-affichage (ab-droit B))

Définition de la fonction

Ainsi, pour implanter la fonction, il suffit :
- à partir de la représentation des sous-arbres gauche et droit (appel récursif),
- de « fabriquer » la liste des lignes de ces deux représentations (en utilisant les fonctions `lignes` et `append`),
- de « maper » une fonction qui ajoute un tiret en tête de ligne sur cette liste de lignes,
- de concaténer la représentation de l'étiquette de la racine de l'arbre donné devant le paragraphe correspondant à la liste obtenue.

Cette expression n'étant pas définie lorsque l'arbre est vide, la définition de la fonction est :

```
(define (ab-affichage B)
  ;; add-tiret-prefixe : Ligne -> Ligne
  ;; (add-tiret-prefixe ligne) rend la ligne obtenue en ajoutant un tiret devant «ligne»
  (define (add-tiret-prefixe ligne)
    (string-append "-" ligne))

  ;; expression de (ab-affichage B):
  (if (ab-vide? B)
      (paragraphe '("vide"))
      (paragraphe-cons
       (->string (ab-etiquette B))
       (paragraphe
        (map add-tiret-prefixe
             (append (lignes (ab-affichage (ab-gauche B)))
                     (lignes (ab-affichage (ab-droit B)))))))))
```

Seconde implantation

Avec la définition précédente, l'évaluation de la fonction `ab-affichage` passe son temps à fabriquer un paragraphe à partir d'une liste de lignes (en utilisant la fonction `paragraphe`) et à refabriquer la liste de lignes à partir de ce paragraphe (en utilisant la fonction `lignes`. Pour éviter cela, on peut définir une fonction auxiliaire, `liste-lignes-affichage`, qui rend la liste des lignes du paragraphe recherché :

```
;;; ab-affichage : ArbreBinaire[α] -> Paragraphe
;;; (ab-affichage B) rend, lorsque «B» est l'arbre vide, le paragraphe constitué de la
;;; seule ligne constituée par le mot qui s'affiche "vide" et, lorsque l'arbre «B» n'est pas
;;; vide, le paragraphe dont la première ligne est l'étiquette de la racine de l'arbre «B»
;;; et dont les lignes suivantes sont égales à cette représentation des deux
;;; sous-arbres de «B», toutes les lignes étant préfixées par un tiret
(define (ab-affichage B)
  ;; add-tiret-prefixe : Ligne -> Ligne
  ;; (add-tiret-prefixe ligne) rend la ligne obtenue en ajoutant un tiret devant «ligne»
  (define (add-tiret-prefixe ligne)
    (string-append "-" ligne))

  ;; liste-lignes-affichage : ArbreBinaire[α] -> LISTE[Ligne]
  ;; (liste-lignes-affichage B) rend (lignes (ab-affichage B))
  (define (liste-lignes-affichage B)
```

```
    (if (ab-vide?    B)
        '("vide")
        (cons
         (->string    (ab-etiquette     B))
         (map  add-tiret-prefixe
               (append   (liste-lignes-affichage       (ab-gauche    B))
                         (liste-lignes-affichage       (ab-droit    B)))))   ) )

  ;; expression de (ab-affichage B):
  (paragraphe     (liste-lignes-affichage        B))  )
```

Troisième implantation

Pour avoir une définition plus efficace[2] de cette fonction, on peut aussi utiliser la technique que nous avons expliquée lorsque nous avons écrit la définition de la fonction `triangle2` : on définit une fonction interne qui rend un paragraphe obtenu en préfixant l'affichage d'un arbre donné par un préfixe donné et on applique cette fonction avec un préfixe vide et l'arbre donné :

```
(define   (ab-affichage     B)
  ;; aff-Aux : Ligne * ArbreBinaire[α] -> LISTE[Ligne]
  ;; (aff-Aux pref B) rend la liste de lignes obtenue en préfixant chaque ligne de
  ;; (lignes (ab-affichage B)) par la chaîne «pref»
... à faire
... à faire
... à faire
... à faire
... à faire
... à faire
... à faire

  ;; expression de (ab-affichage B) :
  (paragraphe     (aff-Aux    ""  B))  )
```

Pour la définition de `aff-Aux` , il suffit de suivre la spécification de `ab-affichage` (le préfixe pour les appels récursifs étant égal à la concaténation du préfixe donné et d'un tiret) :

```
(define   (ab-affichage     B)
  ;; aff-Aux : Ligne * ArbreBinaire[α] -> Paragraphe
  ;; (aff-Aux pref B) rend le paragraphe obtenu en préfixant chaque ligne du paragraphe
  ;; « (ab-affichage B) » par la chaîne «pref»
  (define   (aff-Aux    pref  B)
    (if  (ab-vide?    B)
         (paragraphe     (list   (string-append     pref  "vide")))
         (let  ((pref2   (string-append     pref  "-")))
           (paragraphe-cons      (string-append     pref  (->string    (ab-etiquette     B)))
                                 (paragraphe-append      (aff-Aux    pref2   (ab-gauche    B))
                                                         (aff-Aux    pref2   (ab-droit    B))))   ) ) )

  ;; expression de (ab-affichage B) :
  (aff-Aux    ""  B)  )
```

Remarque (efficacité) : la structure des deux définitions que nous avons données pour cette fonction sont similaires sauf que la première définition exécute systématiquement en plus, pour chaque appel récursif, un « map » sur les lignes de la représentration (rappelons que le temps de l'exécution d'un map est de l'ordre de la longueur de la liste donnée). Ainsi, la seconde définition est bien plus performante que la première.

Pour s'auto-évaluer

[2] Voir la remarque à la fin de la présente section

4. Arbres binaires de recherche

Les arbres binaires sont des structures de données très utiles en informatique. Dans cette section, comme exemple d'utilisation des arbres binaires, nous étudions les « arbres binaires de recherche ».

4.1. Introduction et définition

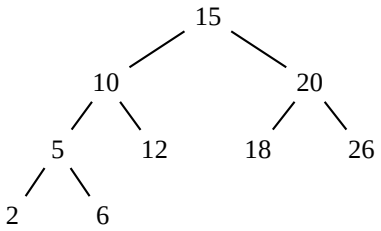
En informatique, il est fréquent d'avoir besoin de gérer un ensemble d'informations (comme l'ensemble des étudiants de DEUG-MIAS). Dans cette gestion, l'une des opérations fondamentales est de savoir si un individu donné (connu par son nom ou par son numéro de carte d'étudiant) fait, ou ne fait pas, partie de la base. Pour ce faire, il existe une structure de données efficace, la structure d'arbre binaire de recherche.

Un arbre binaire de recherche est un arbre binaire vide ou un arbre binaire qui possède les propriétés suivantes :

- l'étiquette de sa racine est
  - supérieure à toutes les étiquettes de son sous-arbre gauche,
  - inférieure à toutes les étiquettes de son sous-arbre droit,
- ses sous-arbres gauche et droit sont aussi des arbres binaires de recherche.

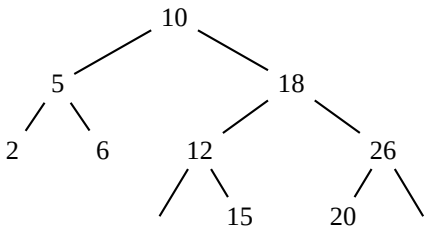
Par la suite, pour que les opérations de comparaison soient définies (et soient < et >), nous considèrerons que les étiquettes sont des nombres.

Voici un exemple d'arbre binaire de recherche :



Les propriétés caractéristiques de la notion d'arbre binaire de recherche sont intéressantes pour l'efficacité de la recherche car elles permettent de ne chercher un élément que dans un des deux sous-arbres, recherche qui s'effectue en ne recherchant que dans un des deux sous-sous-arbres...

Notons que, dans un arbre binaire de recherche, c'est l'ensemble de ses étiquettes qui nous intéresse : deux arbres binaires de recherche peuvent donc être « équivalents ». Par exemple, en tant qu'arbres binaires de recherche, l'arbre précédent est « équivalent » à l'arbre suivant :



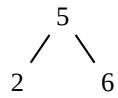
4.2. Spécification

Nous nommerons `ArbreBinRecherche` le type des arbres binaires de recherche, et, par la suite, nous voudrions définir les fonctions suivantes :

```
;;; abr-recherche : Nombre * ArbreBinRecherche -> ArbreBinRecherche + #f
;;; (abr-recherche x ABR) rend l'arbre de racine «x», lorsque «x» apparait dans «ABR»
;;; et renvoie #f si «x» n'apparait pas dans «ABR»
```

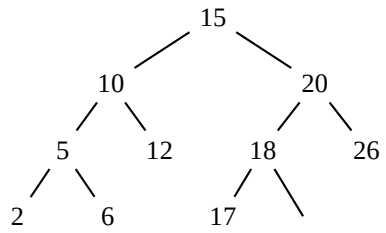
[3] `http://127.0.0.1:20022/q-ab-barriere-arbre-bin-1.quizz`

Par exemple, dans notre implantation, si `abr1` est l'arbre binaire de recherche dessiné ci-dessus, `(abr-recherche 5 abr1)` a comme valeur



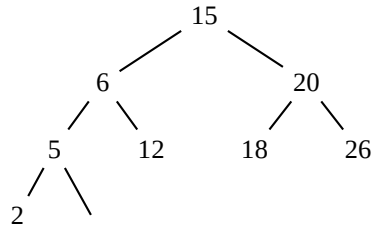

```
;;;  abr-ajout : Nombre * ArbreBinRecherche -> ArbreBinRecherche
;;;  (abr-ajout x ABR) rend l'arbre «ABR» lorque «x» apparait dans «ABR» et, lorsque
;;;  «x» n'apparait pas dans «ABR», rend un arbre binaire de recherche qui contient «x»
;;;  et toutes les étiquettes qui apparaissent dans «ABR»
```

Par exemple, dans notre implantation – et une autre implantation peut rendre un arbre binaire de recherche équivalent –, `(abr-ajout 17 abr1)` a comme valeur




```
;;;  abr-moins : Nombre * ArbreBinRecherche -> ArbreBinRecherche
;;;  (abr-moins x ABR) rend l'arbre «ABR» lorque «x» n'apparait pas dans «ABR» et,
;;;  lorsque «x» apparait dans «ABR», rend un arbre binaire de recherche qui
;;;  contient toutes les étiquettes qui apparaissent dans «ABR» hormis «x».
```

Par exemple, dans notre implantation – et une autre implantation peut rendre un arbre binaire de recherche équivalent –, `(abr-moins 10 abr1)` a comme valeur



Noter que lorsque l'élément à supprimer est l'étiquette d'un nœud interne, obligatoirement, l'arbre binaire est profondemment modifié.

4.3. Implantation

Naturellement, nous implantons les arbres binaires de recherche à l'aide de la barrière d'abstraction `ArbreBinaire`.

4.3.1. Fonction `abr-recherche`

Rappelons la spécification :

```
;;;  abr-recherche : Nombre * ArbreBinRecherche -> ArbreBinRecherche + #f
;;;  (abr-recherche x ABR) rend l'arbre de racine «x», lorsque «x» apparait dans «ABR»
;;;  et renvoie #f si «x» n'apparait pas dans «ABR»
```

L'idée (récursive) de l'implantation est très simple :
- lorsque l'élément recherché est égal à l'étiquette de la racine, on l'a trouvé et le résultat est l'arbre lui-même ;
- sinon, on doit rechercher l'élément dans le sous-arbre gauche ou dans le sous-arbre droit suivant qu'il est inférieur ou supérieur à l'étiquette de la racine.

D'où la définition :

```
(define  (abr-recherche     x  ABR)
  (if  (ab-vide?    ABR)
       #f
       (let  ((e  (ab-etiquette      ABR)))
         (cond  ((=  x  e)  ABR)
                ((<  x  e)  (abr-recherche     x  (ab-gauche    ABR)))
                (else  (abr-recherche     x  (ab-droit    ABR)))))))
```

4.3.2. Fonction `abr-ajout`

Rappelons la spécifi cation :

```
;;;  abr-ajout : Nombre * ArbreBinRecherche -> ArbreBinRecherche
;;;  (abr-ajout x ABR) rend l'arbre «ABR» lorque «x» apparait dans «ABR» et, lorsque
;;;  «x»  n'apparait pas dans «ABR», rend un arbre binaire de recherche qui contient «x»
;;;  et toutes les étiquettes qui apparaissent dans «ABR»
```

L'idée (récursive) de l'implantation est également très simple :

- lorsque l'élément à ajouter est égal à l'étiquette de la racine, le résultat est l'arbre donné ;
- lorsque l'élément à ajouter est inférieur à l'étiquette de la racine, il doit être dans le sous-arbre gauche, aussi le résultat est l'arbre
  - dont l'étiquette est l'étiquette de l'arbre donné,
  - dont le sous-arbre gauche est l'arbre obtenu en ajoutant l'élément dans le sous-arbre gauche de l'arbre donné,
  - dont le sous-arbre droit est le sous-arbre droit de l'arbre donné ;
- lorsque l'élément à ajouter est supérieur à l'étiquette de la racine, il doit être dans le sous-arbre droit et on raisonne comme ci-dessus en inversant sous-arbre gauche et sous-arbre droit.

D'où la défi nition :

```
(define   (abr-ajout    x  ABR)
  (if  (ab-vide?    ABR)
       (ab-noeud    x  (ab-vide)    (ab-vide))
       (let  ((e  (ab-etiquette      ABR)))
         (cond  ((=  x  e)  ABR)
                ((<  x  e)  (ab-noeud    e
                                         (abr-ajout    x  (ab-gauche    ABR))
                                         (ab-droit    ABR)))
                (else  (ab-noeud    e
                                    (ab-gauche    ABR)
                                    (abr-ajout    x  (ab-droit    ABR))))))))
```

Remarque : le recueil d'exercices contient une autre implantation de cette fonction.

4.3.3. Fonction `abr-moins`

Rappelons la spécifi cation :

```
;;;  abr-moins : Nombre * ArbreBinRecherche -> ArbreBinRecherche
;;;  (abr-moins x ABR) rend l'arbre «ABR» lorque «x» n'apparait pas dans «ABR» et,
;;;  lorsque «x»  apparait dans «ABR», rend un arbre binaire de recherche qui
;;;  contient toutes les étiquettes qui apparaissent dans «ABR» hormis «x».
```

L'idée initiale est encore très simple (mais ça se complique par la suite...) :

- lorsque l'élément à supprimer est inférieur à l'étiquette de la racine, cet élément ne peut être que dans le sous-arbre gauche, et le résultat est donc l'arbre
  - dont l'étiquette est l'étiquette de l'arbre donné,
  - dont le sous-arbre gauche est l'arbre obtenu en supprimant l'élément du sous-arbre gauche de l'arbre donné,
  - dont le sous-arbre droit est le sous-arbre droit de l'arbre donné ;
- lorsque l'élément à supprimer est supérieur à l'étiquette de la racine, cet élément ne peut être que dans le sous-arbre droit, et on raisonne comme ci-dessus en inversant sous-arbre gauche et sous-arbre droit ;

— reste le cas où l'élément à supprimer est égal à l'étiquette de la racine ; il faut le supprimer, mais on doit alors complètement reconstruire l'arbre, et de tel sorte que cet arbre vérifie la propriété « être un arbre binaire de recherche » ; compliqué... comme d'habitude, dans ce cas, on utilise et spécifie une nouvelle fonction :

```
(define (abr-moins x ABR)
  (if (ab-vide? ABR)
      (ab-vide)
      (let ((e (ab-etiquette ABR)))
        (cond ((= x e) (moins-racine ABR))
              ((< x e) (ab-noeud e
                                 (abr-moins x (ab-gauche ABR))
                                 (ab-droit ABR)))
              (else (ab-noeud e
                              (ab-gauche ABR)
                              (abr-moins x (ab-droit ABR))))))))
```

La spécification de `moins-racine` étant :

```
;;; moins-racine : ArbreBinRecherche -> ArbreBinRecherche
;;; (moins-racine ABR) rend l'arbre binaire de recherche qui contient toutes
;;; les étiquettes qui apparaissent dans «ABR» hormis l'étiquette de sa racine.
;;; ERREUR lorsque l'arbre «ABR» est vide
```

Comment implanter cette fonction ?

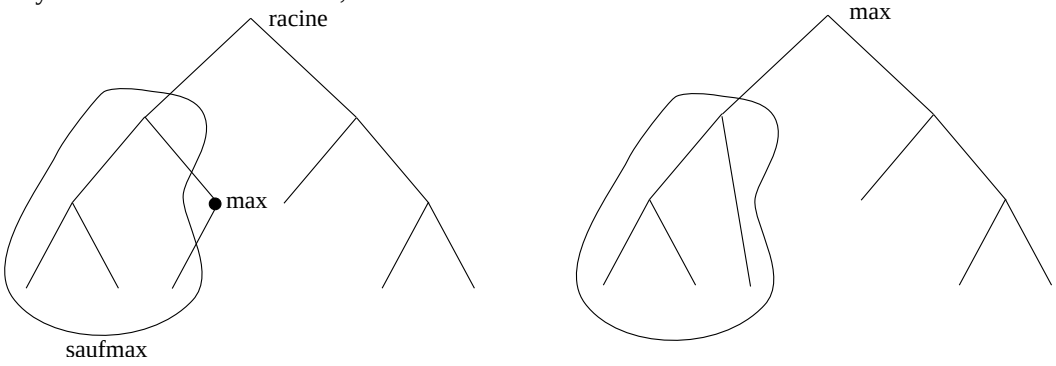
Première idée :

```
(define (moins-racine0 ABR)
  (reduce abr-ajout
          (ab-gauche ABR)
          (ab-liste-infixe (ab-droit ABR))))
```

Seconde idée :

Deux situations entraînent une solution évidente : lorsque l'un des deux sous-arbres est vide, le résultat est l'autre sous-arbre. Dans le cas contraire, on peut essayer de « conserver » un des deux sous-arbres, le droit par exemple : pour obtenir un arbre qui vérifie la propriété « arbre binaire de recherche », il faut alors que la racine de l'arbre résultat soit le plus grand élément du sous-arbre gauche. Ainsi, le résultat est un arbre

- ayant comme racine le plus grand élément du sous-arbre gauche de l'arbre donné (noter que c'est celui qui est au bout de la branche la plus à droite de ce sous-arbre gauche),
- ayant comme sous-arbre gauche, le sous-arbre gauche de l'arbre donné privé de son plus grand élément,
- ayant comme sous-arbre droit, le sous-arbre droit de l'arbre donné :



Ainsi, pour implanter la fonction `moins-racine`, on a besoin de la fonction `max-sauf-max` de spécification :

```
;;; max-sauf-max : ArbreBinRecherche -> NUPLET[Nombre ArbreBinRecherche]
;;; (max-sauf-max ABR) rend le couple formé de la plus grande étiquette présente
;;; dans l'arbre «ABR» et d'un arbre binaire de recherche qui contient toutes les
;;; étiquettes qui apparaissent dans «ABR» hormis ce maximum
```

```
;;; ERREUR lorsque l'arbre «ABR» est vide
```

La définition de la fonction `moins-racine` étant alors :

```
(define (moins-racine ABR)
  (cond ((ab-vide? (ab-gauche ABR))
         (ab-droit ABR))
        ((ab-vide? (ab-droit ABR))
         (ab-gauche ABR))
        (else
         (let ((m-sm-ss-ab-g (max-sauf-max (ab-gauche ABR))))
           (ab-noeud (car m-sm-ss-ab-g)
                     (cadr m-sm-ss-ab-g)
                     (ab-droit ABR)))))))
```

Ne reste plus qu'à implanter la fonction `max-sauf-max`. Comme nous l'avons noté, le plus grand des éléments présents dans un arbre binaire de recherche se trouve au bout de la branche la plus à droite de cet arbre. Ainsi, la recherche du maximum et la constitution de l'arbre privé de son maximum s'effectuent par l'exploration successive des sous-arbres droits :

```
;;; max-sauf-max : ArbreBinRecherche -> NUPLET[Nombre ArbreBinRecherche]
;;; (max-sauf-max ABR) rend le couple formé de la plus grande étiquette présente
;;; dans l'arbre «ABR» et d'un arbre binaire de recherche qui contient toutes les
;;; étiquettes qui apparaissent dans «ABR» hormis ce maximum
;;; ERREUR lorsque l'arbre «ABR» est vide
(define (max-sauf-max ABR)
  (if (ab-vide? (ab-droit ABR))
      (list (ab-etiquette ABR) (ab-gauche ABR))
      (let ((m-sm-ss-ab-d (max-sauf-max (ab-droit ABR))))
        (list (car m-sm-ss-ab-d)
              (ab-noeud (ab-etiquette ABR)
                        (ab-gauche ABR)
                        (cadr m-sm-ss-ab-d))))))
```

5. Implantations des arbres binaires

Dans cette section, pour implanter les arbres binaires, nous étudions les notions Scheme de Sexpressions et de vecteurs.

Ainsi, nous pourrons implanter les arbres binaires de deux façons différentes. Noter que nous ne comparerons pas les performances de ces implantations, une telle étude étant hors du programme de ce cours.

5.1. Notion de Sexpression

Considérons l'expression Scheme, `(- (+ (- 95) 5) (+ 10 3))`. Cela ressemble à une liste (de trois éléments) de type `LISTE[ α]`, le premier élément étant `-`, le second élément étant `(+ (- 95) 5)` et le dernier élément étant `(+ 10 3)` ; à nouveau, `(+ (- 95) 5)` ressemble à une liste de type `LISTE[ α]` (de trois éléments), le premier élément étant `+`, le second élément étant `(- 95)` et le dernier élément étant `5` ; à nouveau, `(- 95)` ressemble à une liste (de deux éléments)... Mais ce ne sont pas des listes de type `LISTE[ α]` puisque les éléments ne sont pas de même type. Ce sont des **Sexpressions**.

5.1.1. Définition

Formellement, on peut définir les Sexpressions par

<Sexpression> → *<atome>*
`LISTE[` *<Sexpression>* `]`

<atome> ::= *<Nombre>*
<bool>
<string>
<Symbole>

(*<Nombre>*, *<bool>*, *<string>* et *<Symbole>* sont définis dans la carte de référence)

5.1.2. Spécification des fonctions primitives

Tout d'abord, nous devons pouvoir savoir si une Sexpression est un atome ou une liste de Sexpressions. Pour ce faire, il existe en Scheme la fonction `list?` :

```
;;;  list?: Valeur -> bool
;;;  (list? v) rend #t ssi v est une liste (éventuellement vide)
```

Les autres fonctions de base sont les fonctions que nous avons déjà vues (`car` , `cdr` , `pair?` ...).

5.1.3. Une implantation des arbres binaires à l'aide des Sexpressions

La notion de Sexpression permet d'implanter efficacement – en Scheme – les arbres binaires (ainsi que tous les types d'arbres). En effet :

- lorsque l'arbre est vide, on peut le représenter par la liste vide (qui est également la Sexpression vide) ;
- lorsque l'arbre n'est pas vide, on peut le représenter par une liste contenant l'étiquette de la racine et les deux Sexpressions qui représentent ses sous-arbres immédiats. Notez que l'on a six ordres possibles (d'abord l'étiquette puis le sous arbre gauche et enfin le sous-arbre droit ou le sous-arbre gauche puis l'étiquette et enfin le sous-arbre droit...). Ces six possibilités sont aussi valables les unes que les autres. Il faut en choisir une et s'en souvenir pour toutes les fonctions qui opèrent sur les arbres non vides. Ci-dessous, nous donnons une implantation écrite en mettant systématiquement l'étiquette de la racine en premier et le sous-arbre gauche avant le sous-arbre droit. La compréhension de cette implantation est facile, aussi nous la donnons telle quelle, sans explication.

```
;;;;  Implantation des arbres binaires à l'aide des Sexpressions. Pour les arbres
;;;;  non vides, on met  systématiquement l'étiquette de la racine en premier et
;;;;  le sous-arbre gauche avant le sous-arbre droit.

;;;;;;;;;;;;;;;        Constructeurs

;;;  ab-noeud : α * ArbreBinaire[α] * ArbreBinaire[α] -> ArbreBinaire[α]
;;;  (ab-noeud e B1 B2) rend l'arbre binaire formé de la racine d'étiquette
;;;  «e», du sous-arbre gauche «B1» et du sous-arbre droit «B2».
(define   (ab-noeud   e B1 B2)
  (list  e B1 B2))

;;;  ab-vide :  -> ArbreBinaire[α]
;;;  (ab-vide) rend l'arbre binaire vide.
(define   (ab-vide)
  '())

;;;;;;;;;;;;;;;        Reconnaisseurs

;;;  ab-noeud? : ArbreBinaire[α] -> bool
;;;  (ab-noeud? B) rend vrai ssi «B» n'est pas l'arbre vide.
(define   (ab-noeud?    B)
  (pair?   B))

;;;  ab-vide? : ArbreBinaire[α] -> bool
;;;  (ab-vide? B) rend vrai ssi «B» est l'arbre vide.
(define   (ab-vide?    B)
  (not  (ab-noeud?    B)))
```

```
;;; ab-etiquette : ArbreBinaire[α] -> α
;;; (ab-etiquette B) rend l'étiquette de la racine de l'arbre «B»
;;; ERREUR lorsque «B» ne satisfait pas ab-noeud?
(define (ab-etiquette B)
  (car B))

;;; ab-gauche : ArbreBinaire[α] -> ArbreBinaire[α]
;;; (ab-gauche B) rend le sous-arbre gauche de «B»
;;; ERREUR lorsque B ne satisfait pas ab-noeud?
(define (ab-gauche B)
  (cadr B))

;;; ab-droit : ArbreBinaire[α] -> ArbreBinaire[α]
;;; (ab-droit B) rend le sous-arbre droit de «B»
;;; ERREUR lorsque «B» ne satisfait pas ab-noeud?
(define (ab-droit B)
  (caddr B))
```

5.2. Notion de Vecteur

Les listes, que nous avons vues dans les cours précédents, permettent de rassembler plusieurs valeurs avec la possibilité d'extraire immédiatement la première valeur (en utilisant `car`) ; mais si l'on veut extraire le $i^{ème}$ élément de la liste (i étant variable), on doit écrire une fonction qui parcourt cette dernière (en utilisant la fonction `cdr`).

Les vecteurs[4] sont des structures de données Scheme qui permettent d'agréger plusieurs valeurs – les composants du vecteur – avec la possibilité d'extraire immédiatement l'une de ces valeurs.

Plus précisément, un vecteur est caractérisé par des **indices**, une **longueur** et des **composants** :

- la longueur du vecteur est le nombre de ses composants,
- ses composants sont indicés par les entiers naturels compris entre 0 (inclus) et sa longueur (exclue) ;
- ainsi, le premier composant du vecteur est celui d'indice 0, le deuxième composant est celui d'indice 1... et le dernier composant est celui ayant comme indice sa longueur moins un.

Notons qu'il existe un vecteur particulier, le vecteur vide, dont la longueur est nulle et qui ne contient aucun composant.

Remarque : sous Drscheme, l'affi chage de la valeur d'un vecteur commence par le caractère #, continue par la longueur du vecteur et se termine par, entre parenthèses, la liste des valeurs des composants du vecteur.

5.2.1. Spécification des fonctions primitives

Constructeur

La fonction `vector` , qui peut avoir un nombre quelconque d'arguments, rend un vecteur dont les composants sont ses arguments, dans l'ordre où ils sont donnés :

```
;;; vector : Valeur ... -> Vecteur
;;; (vector x0 ...) rend le vecteur dont le premier composant (celui d'indice 0) est
;;; «x0», dant le deuxième composant est «x1»...
;;; (vector) rend le vecteur vide (de longueur 0 et qui ne contient aucune valeur)
```

Exemple

```
(vector 'a 'b 'c) →#3(a b c)
```

Accesseurs

```
;;; vector-length : Vecteur -> nat
;;; (vector-length V) rend la longueur de «V», c'est-à-dire son nombre de composants
```

Exemple

[4] On parle de tableaux dans d'autres langages de programmation (avec une nuance que nous ne verrons pas ici).

(vector-length (vector 'a 'b 'c)) →3

Rappelons que le dernier indice d'un vecteur V est égal à (- (vector-length V) 1).

Nous avons dit que la caractéristique d'un vecteur était de pouvoir accéder immédiatement à un composant d'un indice donné. Ceci se fait grâce à la fonction vector-ref :

```
;;; vector-ref : Vecteur * nat -> Valeur
;;; (vector-ref V k) rend le composant d'indice «k» du vecteur «V»
;;; ERREUR lorsque «k» n'appartient pas à l'intervalle [0 .. (vector-length V)[
```

Exemples

```
(vector-ref (vector 'a 'b 'c) 0) →a

(vector-ref (vector 'a 'b 'c) 2) →c
```

(vector-ref (vector 'a 'b 'c) 3) rend une erreur (vector-ref: index 3 out of range [0, 2] for vector: #3(a b c))

Fonctions de conversion

Comme nous l'avons dit, les listes et les vecteurs représentent des suites d'éléments (mais les fonctions de base sont très différentes). Aussi existe-t-il deux fonctions, vector->list et list->vector, qui permettent de passer d'une structure de données à l'autre :

```
;;; vector->list : Vecteur -> LISTE[Valeur]
;;; (vector->list V) rend la liste des composants du vecteur «V»
```

Exemple

```
(vector->list (vector 'a 'b 'c)) →(a b c)
```

```
;;; list->vector : LISTE[Valeur] -> Vecteur
;;; (list->vector L) rend le vecteur ayant comme premier composant le premier
;;; élément de la liste «L», comme deuxième composant le deuxième élément de la liste «L»...
```

Exemple

```
(list->vector '(a b c)) →#3(a b c)
```

5.2.2. Une implantation des arbres binaires à l'aide des vecteurs

On peut implanter les arbres binaires en utilisant des vecteurs :

- l'arbre vide est représenté par le vecteur vide,
- un arbre non-vide est représenté par un vecteur de longueur trois ayant comme composants l'étiquette de la racine de l'arbre – par exemple en indice 0 –, la représentation du sous-arbre gauche – par exemple en indice 1 – et la représentation du sous-arbre droit – par exemple en indice 2.

Constructeurs

La fonction ab-noeud est implantée en utilisant la fonction vector :

```
;;; ab-noeud : α * ArbreBinaire[α] * ArbreBinaire[α] -> ArbreBinaire[α]
;;; (ab-noeud e B1 B2) rend l'arbre binaire formé de la racine d'étiquette
;;; «e», du sous-arbre gauche «B1» et du sous-arbre droit «B2».
(define (ab-noeud e B1 B2)
  (vector e B1 B2))
```

La fonction ab-vide est implantée par le vecteur vide en utilisant aussi la fonction vector (mais sans argument) :

```
;;; ab-vide : -> ArbreBinaire[α]
;;; (ab-vide) rend l'arbre binaire vide.
(define (ab-vide)
  (vector))
```

Reconnaisseurs

Pour savoir si un arbre est, ou n'est pas, vide il suffi t de comparer la longueur du vecteur à 0 :

```
;;; ab-vide? : ArbreBinaire[α] -> bool
;;; (ab-vide? B) rend vrai ssi «B» est l'arbre vide.
(define (ab-vide? B)
  (= (vector-length B) 0))

;;; ab-noeud? : ArbreBinaire[α] -> bool
;;; (ab-noeud? B) rend vrai ssi «B» n'est pas l'arbre vide.
(define (ab-noeud? B)
  (> (vector-length B) 0))
```

Accesseurs

Pour les accesseurs, on utilise la fonction `vector-ref`, l'étiquette étant en indice 0, le sous-arbre gauche en indice 1 et le sous-arbre droit en indice 2 :

```
;;; ab-etiquette : ArbreBinaire[α] -> α
;;; (ab-etiquette B) rend l'étiquette de la racine de l'arbre «B»
;;; ERREUR lorsque «B» ne satisfait pas ab-noeud?
(define (ab-etiquette B)
  (vector-ref B 0))

;;; ab-gauche : ArbreBinaire[α] -> ArbreBinaire[α]
;;; (ab-gauche B) rend le sous-arbre gauche de «B»
;;; ERREUR lorsque B ne satisfait pas ab-noeud?
(define (ab-gauche B)
  (vector-ref B 1))

;;; ab-droit : ArbreBinaire[α] -> ArbreBinaire[α]
;;; (ab-droit B) rend le sous-arbre droit de «B»
;;; ERREUR lorsque «B» ne satisfait pas ab-noeud?
(define (ab-droit B)
  (vector-ref B 2))
```

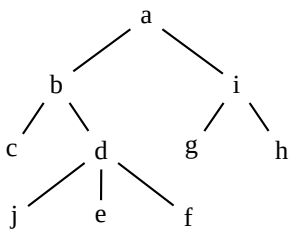
6. Arbres généraux

6.1. Caractéristiques des arbres généraux

Parmi la famille des arbres, les arbres généraux sont caractérisés par :

- information attachée à chaque nœud,
- non existence de l'arbre vide,
- chaque nœud a un nombre quelconque (éventuellement 0) de sous-arbres immédiats.

Voici un exemple d'arbre général :



Lorsque le type des étiquettes attachées aux nœuds est α, on notera `ArbreGeneral[α]` le type des arbres généraux.

Pour les arbres binaires, les sous-arbres immédiats étaient le sous-arbre gauche et le sous-arbre droit. Pour les arbres généraux, comme le nombre de sous-arbres immédiats est quelconque, on ne peut pas les caractériser aussi

Formellement. Ainsi défi nit-on la notion de forêt : une **foret** est une liste d'arbres (généraux). Nous noterons `Foret[ α]` le type des forêts dont les arbres appartiennent à `ArbreGeneral[ α]`.
Ainsi, `Foret[ α]` est le type `LISTE[ArbreGeneral[ α]]`.

Ainsi, les sous-arbres immédiats d'un arbre général constituent une forêt : un arbre général est constitué par une racine, et l'étiquette attachée à cette racine, et par la forêt de ses sous-arbres immédiats.

La forêt des sous-arbres immédiats d'un arbre peut être la liste vide : l'arbre est alors réduit à une *feuille*.

6.2. Barrière d'abstraction des arbres généraux

6.2.1. Constructeur

La barrière d'abstraction des arbres généraux ne comporte qu'un constructeur, la fonction `ag-noeud` :

```
;;; ag-noeud : α * Foret[α] -> ArbreGeneral[α]
;;; avec Foret[α] == LISTE[ArbreGeneral[α]]
;;; (ag-noeud e foret) rend l'arbre formé de la racine d'étiquette «e» et,
;;; comme sous-arbres immédiats, les arbres de la forêt «foret».
```

Exemples

Défi nissons la fonction `ag-feuille` spécifi ée par :

```
;;; ag-feuille : α -> ArbreGeneral[α]
;;; (ag-feuille e) rend l'arbre général réduit à une feuille ayant «e» comme étiquette
```

Pour ce faire, il suffi t de remarquer qu'il s'agit de l'arbre ayant `e` comme étiquette de la racine est ayant la liste vide comme forêt des sous-arbres immédiats :

```
(define (ag-feuille e)
  (ag-noeud e '()))
```

Comme autre exemple, l'arbre dessiné ci-dessus peut être construit avec l'expression :

```
(ag-noeud
 'a
 (list (ag-noeud
        'b
        (list (ag-feuille 'c)
              (ag-noeud
               'd
               (list (ag-feuille 'j)
                     (ag-feuille 'e)
                     (ag-feuille 'f)))))
       (ag-noeud
        'i
        (list (ag-feuille 'g)
              (ag-feuille 'h)))))
```

6.2.2. Affichage

Comme pour les arbres binaires, on ne vous montre rien de la représentation des arbres ainsi construits. Aussi la barrière d'abstraction que nous vous fournissons comporte la fonction `ab-expression` , qui a comme spécifi cation :

```
;;; ag-expression: ArbreGeneral[α] -> Sexpression
;;; (ag-expression g) rend une Sexpression reflétant la construction de l'arbre «g»
```

Exemple

`(ag-expression (ag-feuille 'a))` rend
`(ag-noeud 'a (list))`

6.2.3. Reconnaisseurs

Tout d'abord, revenons sur la notion de reconnaisseur. Lorsque l'on utilise une barrière d'abstraction, les objets manipulés sont (tous) « fabriqués » en utilisant les constructeurs et, lorsque l'on veut défi nir une fonction qui a comme donnée un tel objet, on a besoin de savoir avec quel constructeur il a été fabriqué. C'est le rôle des reconnaisseurs, qui permettent d'aiguiller les défi nitions entre « l'objet a été fabriqué par tel constructeur » ou « l'objet a été fabriqué par tel autre constructeur » ou... Mais alors, lorsque la barrière d'abstraction ne comporte qu'un constructeur, tout objet est « fabriqué » en utilisant ce constructeur et on n'a pas besoin de reconnaisseur.

Ainsi, dans la barrière d'abstraction des arbres généraux, comme il n'y a qu'un constructeur, il n'y a pas de reconnaisseur.

6.2.4. Accesseurs

;;; *ag-etiquette : ArbreGeneral[α] -> α*
;;; *(ag-etiquette g) rend l'étiquette de la racine de l'arbre «g».*

;;; *ag-foret : ArbreGeneral[α] -> Foret[α]*
;;; *avec Foret[α] == LISTE[ArbreGeneral[α]]*
;;; *(ag-foret g) rend la forêt des sous-arbres immédiats de «g».*

Exemple

On peut écrire la défi nition du prédicat `ag-feuille?` spécifi ée par :

;;; *ag-feuille? : ArbreGeneral[α] -> bool*
;;; *(ag-feuille? G) rend #t ssi «G» est un arbre réduit à une feuille*

en utilisant le prédicat `pair?` :

```
(define (ag-feuille? G)
  (not (pair? (ag-foret G))))
```

6.2.5. Propriétés remarquables

Les fonctions de la barrière d'abstraction des arbres généraux vérifi ent les propriétés suivantes :

Pour toute liste d'arbres généraux (ou forêt), `F`, et toute valeur, `v` :

```
(ag-etiquette (ag-noeud v F) ) → v
(ag-foret (ag-noeud v F) ) → F
```

Pour tout arbre général, `G` :

```
(ag-noeud (ag-etiquette G) (ag-foret G) ) → G
```

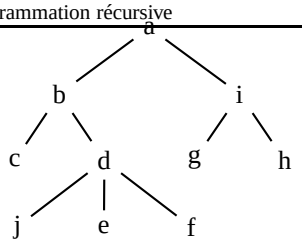
6.3. Exemples d'utilisations de la barrière d'abstraction

Les défi nitions récursives sur les arbres généraux sont plus compliquées que les défi nitions sur les arbres binaires car on ne peut pas atteindre directement tous les sous-arbres immédiats d'un arbre donné. En fait, souvent, pour défi nir une fonction ayant comme donnée un arbre général, nous aurons besoin de défi nir une autre fonction ayant comme donnée une forêt, ces deux fonctions étant mutuellement récursives (l'une appelle l'autre et inversement). On parle alors de récursivité croisée.

6.3.1. Profondeur d'un arbre

Comme nous l'avons fait pour les arbres binaires, nous voudrions calculer la profondeur des arbres généraux. La profondeur d'un arbre peut être défi nie comme étant égale à un de plus que la profondeur de la forêt constituée par ses sous-arbres immédiats, sachant que la profondeur d'une forêt est la profondeur de son arbre le plus profond (et elle est nulle pour la forêt vide).

Par exemple, la profondeur de l'arbre



est égale à

```
4
```

Remarquer que la profondeur d'un arbre est le nombre de niveaux lorsque l'on dessine l'arbre comme ci-dessus. Défi nissons donc la fonction `ag-profondeur` ayant comme spécifi cation

```
;;; ag-profondeur : ArbreGeneral[α] -> nat
;;; (ag-profondeur G) rend la profondeur de l'arbre «G»
```

Comme cela a été dit dans l'introduction, pour défi nir cette fonction, nous avons besoin de défi nir aussi la fonction qui rend la profondeur d'une forêt. Cette fonction n'étant qu'une fonction auxiliaire, nous la défi nissons à l'intérieur de la défi nition de la fonction `ag-profondeur` :

```
(define (ag-profondeur G)
  ;; profondeurForet : Foret[α] -> nat
  ;; (profondeurForet F) rend la profondeur de F, c.-à-d. le maximum des profondeurs
  ;; des arbres de F (rend 0 lorsque la forêt est vide).
... à faire
... à faire
... à faire
... à faire
  ;; expression de (ag-profondeur G) :
... à faire
```

D'après la défi nition de la profondeur d'un arbre, la profondeur de G est égale à un de plus que la profondeur de la forêt de ses sous-arbres immédiats :

```
(define (ag-profondeur G)
  ;; profondeurForet : Foret[α] -> nat
  ;; (profondeurForet F) rend la profondeur de F, c.-à-d. le maximum des profondeurs
  ;; des arbres de F (rend 0 lorsque la forêt est vide).
... à faire
... à faire
... à faire
... à faire
  ;; expression de (ag-profondeur G) :
  (+ 1 (profondeurForet (ag-foret G))))
```

Défi nissons maintenant la fonction qui calcule la profondeur d'une forêt. Une forêt étant une liste, ce n'est qu'un exercice de révisionsur les listes : la profondeur de la forêt est égale au maximum de la profondeur du premier arbre de la forêt et de la profondeur de la forêt obtenue, à partir de la forêt donnée, en supprimant son premier arbre. Cette relation de récurrence n'étant défi nie que pour une liste non vide, il faut extraire de l'appel récursif le cas où `(pair? F)` est faux. D'où la défi nition :

```
(define (ag-profondeur G)
  ;; profondeurForet : Foret[α] -> nat
  ;; (profondeurForet F) rend la profondeur de F, c.-à-d. le maximum des profondeurs
  ;; des arbres de F (rend 0 lorsque la forêt est vide).
  (define (profondeurForet F)
```

```
    (if (pair? F)
        (max (ag-profondeur (car F)) (profondeurForet (cdr F)))
        0))
  ;; expression de (ag-profondeur G) :
  (+ 1 (profondeurForet (ag-foret G))))
```

Autre définition

Pour la définition de la fonction `profondeurForet`, au lieu d'écrire explicitement la récursivité comme nous l'avons fait ci-dessus, on peut utiliser les fonctionnelles sur les listes.

Nous avons besoin de la profondeur de tous les arbres de la forêt, ce que nous pouvons calculer en utilisant la fonctionnelle `map` – appliquée à la fonction `ag-profondeur` et à `F` et qui rend la liste des profondeurs des arbres. Ensuite, nous devons calculer le maximum des éléments de cette liste, ce que l'on peut faire en appliquant la fonctionnelle `reduce` à cette liste, à la fonction `max` et en prenant `0` comme valeur initiale :

```
;;; ag-profondeur : ArbreGeneral[α] -> nat
;;; (ag-profondeur G) rend la profondeur de l'arbre «G»
(define (ag-profondeur G)
  ;; profondeurForet : Foret[α] -> nat
  ;; (profondeurForet F) rend la profondeur de «F», c.-à-d. le maximum des profondeurs
  ;; des arbres de «F» (rend 0 lorsque la foret est vide).
  (define (profondeurForet F)
    (reduce max 0 (map ag-profondeur F)))
  ;; expression de (ag-profondeur G) :
  (+ 1 (profondeurForet (ag-foret G))))
```

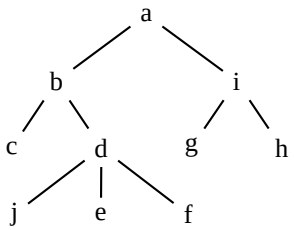
Une définition encore plus concise

En fait, en utilisant les fonctionnelles `map` et `reduce`, nous n'avons pas besoin de définir explicitement la fonction `profondeurForet` : dans la définition de `ag-profondeur` il suffit de substituer son appel par son expression (en n'oubliant pas les arguments de l'appel) :

```
;;; ag-profondeur : ArbreGeneral[α] -> nat
;;; (ag-profondeur G) rend la profondeur de l'arbre «G»
(define (ag-profondeur G)
  (+ 1 (reduce max 0 (map ag-profondeur (ag-foret G)))))
```

6.3.2. Liste préfixe des étiquettes d'un arbre

La liste préfixe des étiquettes d'un arbre général est égale à la concaténation de l'étiquette de sa racine et de la liste préfixe des étiquettes de chacun de ses sous-arbres immédiats. Par exemple, la liste préfixe de l'arbre



est égale à

```
(a b c d j e f i g h)
```

Cherchons une définition de la fonction `ag-liste-prefixe` de spécification :

```
;;; ag-liste-prefixe : ArbreGeneral[α] -> LISTE[α]
;;; (ag-liste-prefixe G) rend la liste préfixe des étiquettes de l'arbre «G»
```

Pour définir cette fonction, comme d'habitude, nous utilisons une fonction qui « fait le travail » pour la forêt de ses sous-arbres immédiats :

```
(define (ag-liste-prefixe G)
  ;; liste-prefixe-foret : Foret[α] -> LISTE[α] rend la concaténation
  ;; des listes préfixes des étiquettes des arbres de F
... à faire
... à faire
... à faire
... à faire
  ;; expression de (ag-liste-prefixe G) :
  (cons (ag-etiquette G) (liste-prefixe-foret (ag-foret G))))
```

La fonction `liste-prefixe-foret` se définit comme d'habitude pour les fonctions sur les listes/

```
(define (ag-liste-prefixe G)
  ;; liste-prefixe-foret : Foret[α] -> LISTE[α] rend la concaténation
  ;; des listes préfixes des étiquettes des arbres de F
  (define (liste-prefixe-foret F)
    (if (pair? F)
        (append (ag-liste-prefixe (car F))
                (liste-prefixe-foret (cdr F)))
        '()))
  ;; expression de (ag-liste-prefixe G) :
  (cons (ag-etiquette G) (liste-prefixe-foret (ag-foret G))))
```

Autre définition

Encore une fois, on peut aussi utiliser les fonctionnelles habituelles sur les listes :

```
;;; ag-liste-prefixe : ArbreGeneral[α] -> LISTE[α]
;;; (ag-liste-prefixe G) rend la liste préfixe des étiquettes de l'arbre «G»
(define (ag-liste-prefixe G)
  ;; liste-prefixe-foret : Foret[α] -> LISTE[α] rend la concaténation
  ;; des listes préfixes des étiquettes des arbres de «F»
  (define (liste-prefixe-foret F)
    (reduce append '() (map ag-liste-prefixe F)))
  ;; expression de (ag-liste-prefixe G) :
  (cons (ag-etiquette G) (liste-prefixe-foret (ag-foret G))))
```

Une définition encore plus concise

Et nous n'avons pas besoin de définir explicitement la fonction `liste-prefixe-foret` :

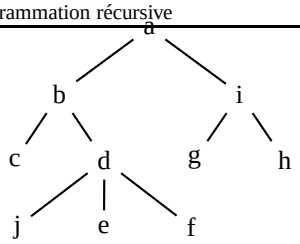
```
;;; ag-liste-prefixe : ArbreGeneral[α] -> LISTE[α]
;;; (ag-liste-prefixe G) rend la liste préfixe des étiquettes de l'arbre «G»
(define (ag-liste-prefixe G)
  (cons (ag-etiquette G)
        (reduce append
                '()
                (map ag-liste-prefixe (ag-foret G)))))
```

6.3.3. Affichage d'un arbre

Nous voudrions afficher les arbres généraux comme nous l'avons fait pour les arbres binaires. Autrement dit, nous voudrions définir la fonction `ag-affichage` qui, étant donné un arbre général, rend un paragraphe

- dont la première ligne est égale à l'étiquette de la racine de l'arbre
- et dont les lignes suivantes sont obtenues en préfixant par un tiret chaque ligne de la représentation ainsi définie de la suite des sous-arbres de l'arbre.

Par exemple `ag-affichage` appliquée à l'arbre



donne

```
"
a
-b
--c
--d
---j
---e
---f
-i
--g
--h
"
```

Spécification de la fonction

Ainsi, la fonction `ag-affichage` a comme spécifi cation :

```
;;; ag-affichage : ArbreGeneral[α] -> Paragraphe
;;; (ag-affichage G) rend le paragraphe dont la première ligne est l'étiquette de
;;; la racine de l'arbre «G» et dont les lignes suivantes sont égales à la
;;; représentation de la suite des sous-arbres de «G», toutes les lignes étant
;;; préfixées par un tiret
```

Une première implantation

Idée

Comme pour les arbres binaires, la première ligne est la représentation de l'étiquette de l'arbre et les lignes suivantes sont obtenues en préfi xant chaque ligne de la concaténation des représentations des sous-arbres par un tiret :

```
a
-b
--c
--d
---j
---e
---f
-i
--g
--h
```

Définition de la fonction

Comme pour les arbres binaires, nous défi nissons une fonction auxiliaire qui rend la liste des lignes du paragraphe résultat. La défi nition est alors (nous vous demandons de bien l'étudier) :

```
(define (ag-affichage G)
  ;; add-tiret-prefixe : Ligne -> Ligne
  ;; (add-tiret-prefixe ligne) rend la ligne obtenue en ajoutant un tiret devant «ligne»
  (define (add-tiret-prefixe ligne)
    (string-append "-" ligne) )
```

```
  ;; liste-lignes-affichage : ArbreGeneral[α] -> LISTE[ligne]
  ;; (liste-lignes-affichage G) rend (lignes (ag-affichage G))
  (define (liste-lignes-affichage G)
    (cons (->string (ag-etiquette G))
          (map add-tiret-prefixe
               (reduce append
                       '()
                       (map liste-lignes-affichage
                            (ag-foret G))))))

  ;; expression de (ag-affichage G):
  (paragraphe (liste-lignes-affichage G)))
```

Une seconde implantation

Pour écrire une définition plus efficace de cette fonction, comme nous l'avons fait pour les arbres binaires, nous pouvons utiliser une fonction auxiliaire, `aff-arbre`, qui affiche un arbre donné en préfixant chaque ligne par une chaîne donnée :

```
(define (ag-affichage G)
  ;; aff-arbre : Ligne * ArbreGeneral[α] -> Paragraphe
  ;; (aff-arbre pref G) rend le paragraphe obtenu en préfixant chaque ligne de
  ;; (ag-affichage G) par la chaîne «pref»
... à faire
... à faire
... à faire
... à faire
... à faire
... à faire
... à faire
... à faire
... à faire
... à faire
... à faire
... à faire
... à faire
... à faire
  ;; expression de (ag-affichage G)
  (aff-arbre "" G))
```

Pour un arbre et un préfixe donnés, la fonction `aff-arbre` rend le paragraphe dont la première ligne est égale à l'image de l'étiquette de la racine de l'arbre donné précédée du préfixe donné et dont les lignes suivantes sont égales à la représentation de la forêt des sous-arbres de l'arbre donné préfixées par le préfixe donné et un nouveau tiret. En utilisant la nouvelle fonction auxiliaire `aff-foret`, spécifiée ci-dessous, la fonction `aff-arbre` peut être définie par :

```
(define (ag-affichage G)
  ;; aff-arbre : Ligne * ArbreGeneral[α] -> Paragraphe
  ;; (aff-arbre pref G) rend le paragraphe obtenu en préfixant chaque ligne de
  ;; (ag-affichage G) par la chaîne «pref»
  (define (aff-arbre pref G)
    (paragraphe-cons (string-append pref (->string (ag-etiquette G)))
                     (aff-foret (string-append pref "-") (ag-foret G))))
  ;; aff-foret : Ligne * ArbreGeneral[α] -> Paragraphe
  ;; (aff-foret pref F) rend le paragraphe obtenu en préfixant chaque ligne
  ;; de la représentation des arbres de «F» par la chaîne «pref»
... à faire
... à faire
```

```
... à faire
... à faire
... à faire
... à faire
... à faire
  ;; expression de (ag-affichage G)
  (aff-arbre "" G))
```

Reste à définir la fonction `aff-foret`. Donnons directement une définition qui utilise les fonctionnelles. Que doit-on faire ? Il faut concaténer tous les éléments de la liste des paragraphes représentant chacun des arbres de la forêt donnée, chaque ligne étant préfixée par le préfixe donné. Pour fabriquer cette liste, nous utilisons la fonctionnelle `map`, appliquée à la fonction, `aff-arbre-pref`, qui affiche un arbre en préfixant chaque ligne par le préfixe donné ; cette fonction doit être définie à l'intérieur de la définition de `aff-foret`, la variable contenant le préfixe étant globale dans cette fonction :

```
(define (ag-affichage G)
  ;; aff-arbre : Ligne * ArbreGeneral[α] -> Paragraphe
  ;; (aff-arbre pref G) rend le paragraphe obtenu en préfixant chaque ligne de
  ;; (ag-affichage G) par la chaîne «pref»
  (define (aff-arbre pref G)
    (paragraphe-cons (string-append pref (->string (ag-etiquette G)))
                     (aff-foret (string-append pref "-") (ag-foret G))))
  ;; aff-foret : Ligne * ArbreGeneral[α] -> Paragraphe
  ;; (aff-foret pref F) rend le paragraphe obtenu en préfixant chaque ligne
  ;; de la représentation des arbres de «F» par la chaîne «pref»
  (define (aff-foret pref F)
    ; aff-arbre-pref : ArbreGeneral[α] -> Paragraphe
    ; (aff-arbre-pref G) rend le paragraphe obtenu en préfixant chaque ligne de
    ; (ag-affichage G) par la chaîne «pref»
... à faire
... à faire
    ; expression de (aff-foret pref F)
    (reduce paragraphe-append (paragraphe '()) (map aff-arbre-pref F)))
  ;; expression de (ag-affichage G)
  (aff-arbre "" G))
```

Pour finir, la définition de la fonction `aff-arbre-pref` n'est qu'une application de la fonction `aff-arbre` :

```
(define (ag-affichage G)
  ;; aff-arbre : Ligne * ArbreGeneral[α] -> Paragraphe
  ;; (aff-arbre pref G) rend le paragraphe obtenu en préfixant chaque ligne de
  ;; (ag-affichage G) par la chaîne «pref»
  (define (aff-arbre pref G)
    (paragraphe-cons (string-append pref (->string (ag-etiquette G)))
                     (aff-foret (string-append pref "-") (ag-foret G))))
  ;; aff-foret : Ligne * ArbreGeneral[α] -> Paragraphe
  ;; (aff-foret pref F) rend le paragraphe obtenu en préfixant chaque ligne
  ;; de la représentation des arbres de «F» par la chaîne «pref»
  (define (aff-foret pref F)
    ; aff-arbre-pref : ArbreGeneral[α] -> Paragraphe
    ; (aff-arbre-pref G) rend le paragraphe obtenu en préfixant chaque ligne de
    ; (ag-affichage G) par la chaîne «pref»
    (define (aff-arbre-pref G)
      (aff-arbre pref G))
    ; expression de (aff-foret pref F)
    (reduce paragraphe-append (paragraphe '()) (map aff-arbre-pref F)))
```

```
  ;; expression de (ag-affichage G)
  (aff-arbre "" G))
```

Une définition plus concise

Encore une fois, nous n'avons pas besoin de défi nir explicitement la fonction sur les forêts :

```
(define (ag-affichage G)
  ;; aff-arbre : Ligne * ArbreGeneral[α] -> Paragraphe
  ;; (aff-arbre pref G) rend le paragraphe obtenu en préfixant chaque ligne de
  ;; (ag-affichage G) par la chaîne «pref»
  (define (aff-arbre pref G)
    (let ((pref2 (string-append pref "-")))
      ; aff-arbre-pref2 : ArbreGeneral[α] -> Paragraphe
      ; (aff-arbre-pref2 G) rend le paragraphe obtenu en préfixant chaque
      ; ligne de (ag-affichage G) par la chaîne «pref2»
      (define (aff-arbre-pref2 G)
        (aff-arbre pref2 G))

      ; expression de (aff-arbre pref G)
      (paragraphe-cons (string-append pref (->string (ag-etiquette G)))
                       (reduce paragraphe-append
                               (paragraphe '())
                               (map aff-arbre-pref2 (ag-foret G))))))

  ;; expression de (ag-affichage G)
  (aff-arbre "" G))
```

6.4. Implantation des arbres généraux

Comme pour les arbres binaires, nous allons montrer que pour une barrière d'abstraction il peut y avoir différentes implantations : ici nous allons en voir trois, l'une à l'aide des Sexpressions, une autre à l'aide des vecteurs et la troisième à l'aide des vecteurs et des Sexpressions.

6.4.1. À l'aide des Sexpressions

Dans cette première implantation, nous représentons les arbres généraux à l'aide de Sexpressions : un arbre général est représenté par une liste dont le premier élément est l'étiquette de la racine et dont les éléments suivants constituent la forêt de ses sous-arbres immédiats.

Définition de la fonction `ag-noeud`

Rappelons sa spécifi cation :

```
;;; ag-noeud : α * Foret[α] -> ArbreGeneral[α]
;;; avec Foret[α] == LISTE[ArbreGeneral[α]]
;;; (ag-noeud e F) rend l'arbre formé de la racine d'étiquette «e» et, comme
;;; sous-arbres, les arbres de la forêt «F».
```

Puisque nous avons décidé que nous représenterions un arbre par une Sexpression dont le premier élément est l'étiquette de la racine et le reste de la liste est la forêt de ses sous-arbres, pour construire un arbre à partir de l'étiquette `e` de la racine et de la forêt `F` de ses sous-arbres, il suffi t d'utiliser la fonction `cons` :

```
(define (ag-noeud e F)
  (cons e F))
```

Définition de la fonction `ag-etiquette`

Rappelons sa spécifi cation :

```
;;; ag-etiquette : ArbreGeneral[α] -> α
;;; (ag-etiquette g) rend l'étiquette de la racine de l'arbre «g».
```

Nous avons dit que l'étiquette était le premier élément de la liste qui représente l'arbre :

```
(define   (ag-etiquette     g)
  (car  g))
```

Définition de la fonction `ag-foret`

Rappelons sa spécification :

```
;;;  ag-foret : ArbreGeneral[α] -> Foret[α]
;;;  avec Foret[α] == LISTE[ArbreGeneral[α]]
;;;  (ag-foret g) rend la forêt des sous-arbres immédiats de «g».
```

Nous avons dit que la forêt des sous-arbres était constituée par la liste qui représente l'arbre hormis son premier élément :

```
(define   (ag-foret    g)
  (cdr  g))
```

6.4.2. À l'aide des vecteurs

Implantons maintenant les arbres généraux en utilisant des vecteurs : un arbre général est représenté par un vecteur, le premier composant du vecteur contenant l'étiquette de la racine de l'arbre et les composants suivants contenant les représentations des sous-arbres immédiats de l'arbre.

Définition de la fonction `ag-noeud`

Rappelons sa spécification :

```
;;;  ag-noeud : α * Foret[α] -> ArbreGeneral[α]
;;;  avec Foret[α] == LISTE[ArbreGeneral[α]]
;;;  (ag-noeud e F) rend l'arbre formé de la racine d'étiquette «e» et, comme
;;;  sous-arbres, les arbres de la forêt «F».
```

Nous devons rendre un vecteur dont nous connaissons le premier composant et la liste des autres composants. Pour ce faire, on peut utiliser la fonction `list->vector` :

```
(define   (ag-noeud    e F)
  (list->vector     (cons  e F)))
```

Définition de la fonction `ag-etiquette`

Rappelons sa spécification :

```
;;;  ag-etiquette : ArbreGeneral[α] -> α
;;;  (ag-etiquette g) rend l'étiquette de la racine de l'arbre «g».
```

L'étiquette étant le premier composant (celui d'indice 0) du vecteur qui représente l'arbre, il suffit d'appliquer la fonction `vector-ref` :

```
(define   (ag-etiquette     g)
  (vector-ref     g 0))
```

Définition de la fonction `ag-foret`

Rappelons sa spécification :

```
;;;  ag-foret : ArbreGeneral[α] -> Foret[α]
;;;  avec Foret[α] == LISTE[ArbreGeneral[α]]
;;;  (ag-foret g) rend la forêt des sous-arbres immédiats de «g».
```

La forêt des sous-arbres étant la liste des arbres contenu dans les composants (hormis le premier) du vecteur, nous utilisons la fonction `vector->list` :

```
(define   (ag-foret    g)
  (cdr  (vector->list     g)))
```

6.4.3. A l'aide des vecteurs et des Sexpressions

Dans cette implantation, nous représentons un arbre général par un vecteur, de longueur deux, dont le premier composant est l'étiquette de la racine et dont le second composant est la liste de ses sous-arbres immédiats (c'est donc la forêt de ses sous-arbres immédiats).

Définition de la fonction `ag-noeud`

Rappelons sa spécification :

```
;;; ag-noeud : α * Foret[α] -> ArbreGeneral[α]
;;; avec Foret[α] == LISTE[ArbreGeneral[α]]
;;; (ag-noeud e F) rend l'arbre formé de la racine d'étiquette «e» et, comme
;;; sous-arbres, les arbres de la forêt «F».
```

D'après la spécification de l'implantation des arbres donnée ci-dessus, cette fonction doit rendre un vecteur dont le premier composant est l'étiquette donnée et dont le second composant est la forêt donnée :

```
(define (ag-noeud e F)
  (vector e F))
```

Définition de la fonction `ag-etiquette`

Rappelons sa spécification :

```
;;; ag-etiquette : ArbreGeneral[α] -> α
;;; (ag-etiquette g) rend l'étiquette de la racine de l'arbre «g».
```

Il suffit de rendre le premier composant du vecteur (celui d'indice 0) :

```
(define (ag-etiquette g)
  (vector-ref g 0))
```

Définition de la fonction `ag-foret`

Rappelons sa spécification :

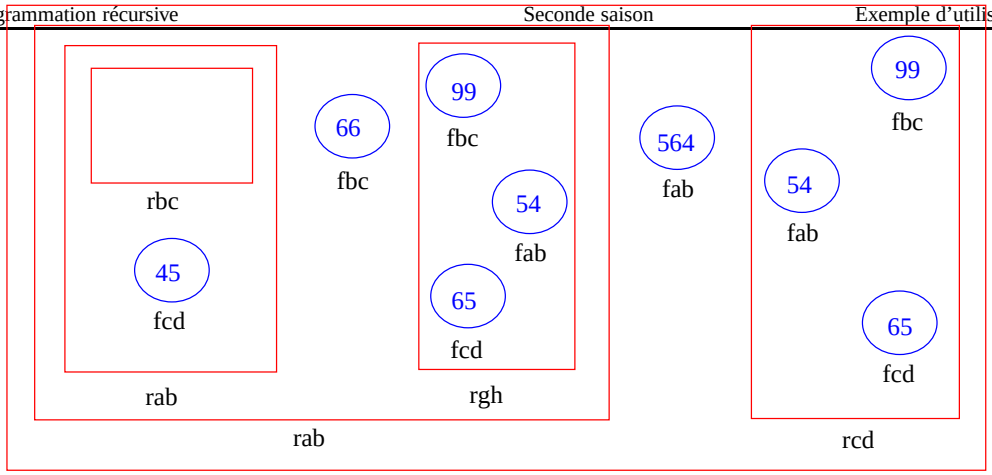
```
;;; ag-foret : ArbreGeneral[α] -> Foret[α]
;;; avec Foret[α] == LISTE[ArbreGeneral[α]]
;;; (ag-foret g) rend la forêt des sous-arbres immédiats de «g».
```

Il suffit de rendre le second composant du vecteur (celui d'indice 1) :

```
(define (ag-foret g)
  (vector-ref g 1))
```

7. Exemple d'utilisation des arbres généraux

Les arbres généraux permettent de représenter efficacement un système de fichiers comme il en existe sous Linux ou sous Windows. En effet, dans un tel système, des répertoires contiennent des fichiers et des répertoires qui contiennent à leur tour des fichiers et des répertoires qui...



7.1. Notion de descripteur de fichier

Dans un tel système, fichiers et répertoires sont décrits à l'aide de **descripteurs** qui, dans la réalité, comportent le nom du fichier ou du répertoire, la date de création, la date de modification... ainsi que des informations pour situer le fichier sur le disque. Dans notre modèle, le descripteur ne comportera que les informations suivantes :

- le nom du fichier ou du répertoire,
- est-ce un fichier ou un répertoire ?,
- pour un fichier, la taille du fichier.

Barrière d'abstraction `Descripteur`

Nous manipulerons les descripteurs à l'aide des fonctions suivantes :

```
;;;;;;;;    Constructeurs
;;;  fichier : string * nat -> Descripteur
;;;  (fichier nom taille) rend le descripteur du fichier de nom «nom» et
;;;  de taille «taille»

;;;  repertoire : string -> Descripteur
;;;  (repertoire nom) rend le descripteur du répertoire de nom «nom»


;;;;;;;;    Reconnaisseurs
;;;  fichier? : Descripteur -> bool
;;;  (fichier? desc) rend #t ssi «desc» est le descripteur d'un fichier

;;;  repertoire? : Descripteur -> bool
;;;  (repertoire? desc) rend #t ssi «desc» est le descripteur d'un répertoire


;;;;;;;;    Accesseurs
;;;  nom :  Descripteur -> string
;;;  (nom desc) rend le nom du répertoire ou du fichier dont le descripteur est «desc»

;;;  taille : Descripteur -> nat
;;;  ERREUR lorsque la description donnée est la description d'un répertoire
;;;  (taille desc) rend la taille du fichier dont le descripteur est «desc»
```

Implantation de la barrière d'abstraction `Descripteur`

On peut implanter cette barrière d'abstraction en utilisant des vecteurs (de longueur deux pour les répertoires et 3 pour les fichiers) :

- le premier composant contient 'D (lorsque c'est un répertoire) ou 'F (lorsque c'est un fichier),

– le second composant contient le nom du répertoire ou du fichier,
– pour les fichiers, le troisième composant contient la taille du fichier.

Remarque : dans notre exemple, les nombres d'informations à mémoriser étant différents pour les fichiers et pour les répertoires, le premier composant du vecteur est inutile. Nous avons tout de même préféré la solution donnée ci-dessus, car, dans le futur, on pourrait être amené à ajouter, dans notre modèle, une autre information pour les répertoires (par exemple, le nombre d'éléments qu'il contient).

L'implantation est alors très simple et nous ne la commenterons pas :

```
;;;;;;;;  Constructeurs
;;; fichier : string * nat -> Descripteur
;;; (fichier nom taille) rend le descripteur du fichier de nom «nom» et
;;; de taille «taille»
(define (fichier nom taille)
  (vector 'F nom taille))
;;; repertoire : string -> Descripteur
;;; (repertoire nom) rend le descripteur du répertoire de nom «nom»
(define (repertoire nom)
  (vector 'D nom))
;;;;;;;;  Reconnaisseurs
;;; fichier? : Descripteur -> bool
;;; (fichier? desc) rend #t ssi «desc» est le descripteur d'un fichier
(define (fichier? desc)
  (equal? (vector-ref desc 0) 'F))
;;; repertoire? : Descripteur -> bool
;;; (repertoire? desc) rend #t ssi «desc» est le descripteur d'un répertoire
(define (repertoire? desc)
  (equal? (vector-ref desc 0) 'D))
;;;;;;;;  Accesseurs
;;; nom : Descripteur -> string
;;; (nom desc) rend le nom du répertoire ou du fichier dont le descripteur est «desc»
(define (nom desc)
  (vector-ref desc 1))
;;; taille : Descripteur -> nat
;;; ERREUR lorsque la description donnée est la description d'un répertoire
;;; (taille desc) rend la taille du fichier dont le descripteur est «desc»
(define (taille desc)
  (vector-ref desc 2))
```

7.2. Représentation d'un système de fichiers

Un système de fichiers peut être représenté par un arbre général dont les étiquettes sont des descripteurs.

Noter que tout arbre ainsi défini ne représente pas un système de fichiers valide : dans un système de fichiers réel, tout nœud qui a comme étiquette un descripteur de fichier doit être une feuille et les noms des différents éléments d'un répertoire doivent être tous différents. Dans la suite, nous nommerons `Systeme` le type des éléments de `ArbreGeneral[Descripteur]` qui vérifient ces deux propriétés.

7.3. Définition de la fonction `du-s`

Nous voudrions définir la fonction `du-s` qui correspond à la commande `du -s` d'Unix :

```
(du-s (syst1))  → 1111

;;; du-s : Systeme -> nat
;;; (du-s systeme) rend la quantité d'espace disque utilisée par les fichiers du
;;; système «systeme»
```

Autrement dit, `(du-s systeme)` rend la somme des tailles de tous les fichiers présents dans le système. La définition de cette fonction est une définition « classique » pour les arbres généraux :

– soit l'étiquette de l'arbre est le descripteur d'un fichier et le résultat est la taille de ce fichier,
– soit c'est un répertoire et il faut sommer les quantités d'espace disque utilisées par les différents éléments du répertoire, cette sommation pouvant être effectuée en utilisant la fonctionnelle `map` :

```
(define   (du-s   systeme)
  (if  (fichier?    (ag-etiquette     systeme))
       (taille   (ag-etiquette     systeme))
       (reduce   + 0 (map   du-s   (ag-foret    systeme)))))
```

7.4. Définition de la fonction `ll`

Nous voudrions définir la fonction `ll` qui correspond à la commande `ls -l` d'Unix :

```
(ll   (syst1))   →
"
0         rab/
0         rgh/
66        fbc
"
```

```
;;;  ll : Systeme -> Paragraphe
;;;  (ll systeme) rend le paragraphe contenant, pour chaque élément de «systeme»
;;;  (on ne considère que les sous-éléments immédiats), une ligne formée:
;;;  pour un fichier, de sa taille et de son nom (séparés par une tabulation),
;;;  pour un répertoire, de 0, d'une tabulation, de son nom immédiatement suivi
;;;  du caractère "/"
```

Pour calculer le résultat de cette fonction, on peut :

1. extraire la liste des descripteurs de tous les sous-arbres du système (en utilisant la fonctionnelle `map` appliquée à la forêt des sous-répertoires et à la fonction `ag-etiquette`),
2. fabriquer la liste des lignes du paragraphe recherché (toujours à l'aide de la fonctionnelle `map` , appliquée à la liste des descripteurs obtenue dans le point précédent et à une fonction – à définir – qui rend la ligne pour un descripteur donné),
3. et il n'y a plus qu'à fabriquer le paragraphe de toutes ces lignes.

Voici la définition Scheme correspondante :

```
(define   (ll  systeme)
  ;;  desc-ll : Descripteur -> Ligne
  ;;  (desc-ll descripteur) rend l'image de «descripteur»: ligne formée:
  ;;  pour un fichier, de sa taille et de son nom (séparés par une tabulation),
  ;;  pour un répertoire, de 0, d'une tabulation, de son nom immédiatement suivi
  ;;  du caractère "/"
  (define   (desc-ll   descripteur)
    (string-append
     (->string   (if  (fichier?   descripteur)
                      (taille   descripteur)
                      0))
     (string   #\tab)
     (nom  descripteur)
     (if  (fichier?   descripteur)    ""  "/")))
  ;;  expression de (ll systeme) :
  (paragraphe    (map  desc-ll
                       (map  ag-etiquette    (ag-foret    systeme)))))
```

7.3. Définition de la fonction `find`

Nous voudrions définir la fonction `find` qui correspond à la commande Unix de même nom :

```
;;; find : string * Systeme -> Paragraphe
;;; (find ident systeme) rend le paragraphe dont les lignes sont constituées par les noms
;;; complets des fichiers ou répertoires du système «systeme» dont le nom est «ident».
```

Voici un exemple de résultat de cette fonction :

```
(find "fbc" (syst1)) →
"
./rab/rgh/fbc
./rab/fbc
./rcd/fbc
"
```

Comme il faut les noms complets, on doit définir une fonction auxiliaire qui a un argument de plus, une chaîne de caractères qui sera un préfixe des lignes de son résultat. De plus, comme d'habitude, pour les définitions de fonctions ayant un arbre généralisé comme donnée, il faut également définir une fonction qui a une forêt comme donnée. D'autre part, on peut noter que la valeur de l'argument « ident » (le nom à chercher) sera inchangée pour tous les appels récursifs : dans les deux fonctions auxiliaires précédentes, cet argument peut être mis en variable globale. D'où la structure de la définition de la fonction `find` :

```
(define (find ident systeme)
  ;; findAux : string * Systeme -> Paragraphe
  ;; (findAux path systeme) rend le paragraphe dont les lignes sont constituées par les
  ;; noms complets des fichiers ou répertoires du système «systeme» dont le nom est
  ;; «ident», chaque ligne étant préfixée par «path».
  ... à faire
  ... à faire
  ;; find-foret : string * Foret[Descripteur] -> Paragraphe
  ;; (find-foret path F) rend le paragraphe obtenu en concaténant, pour tout arbre «A»
  ;; de la fôret «F», (findAux path A)
  ... à faire
  ... à faire
  ;; expression de (find ident systeme) :
  (findAux "" systeme))
```

Définition de la fonction `findAux`

Le résultat de l'application de `findAux` est composé

- d'une ligne comportant le nom complet du système donné lorsque le nom de ce système est le nom recherché,
- du résultat de la recherche sur la forêt du contenu du système donné lorsque ce système est un répertoire.

D'où la définition :

```
(define (find ident systeme)
  ;; findAux : string * Systeme -> Paragraphe
  ;; (findAux path systeme) rend le paragraphe dont les lignes sont constituées par les
  ;; noms complets des fichiers ou répertoires du système «systeme» dont le nom est
  ;; «ident», chaque ligne étant préfixée par «path».
  (define (findAux path systeme)
    (paragraphe-append
     (if (equal? ident (nom (ag-etiquette systeme)))
         (paragraphe (list (string-append path ident)))
         (paragraphe '()))
     (if (repertoire? (ag-etiquette systeme))
         (find-foret (string-append path (nom (ag-etiquette systeme)) "/")
                     (ag-foret systeme))
         (paragraphe '()))))
```

```
;; find-foret : String * Foret[Descripteur] -> Paragraphe
;; (find-foret path F) rend le paragraphe obtenu en concaténant, pour tout arbre «A»
;; de la fôret «F», (findAux path A)
...
```

Définition de la fonction `find-foret`

Nous pouvons utiliser les fonctionnelles `map` et `reduce` , `map` permettant de fabriquer la liste des paragraphes résultats de la recherche sur les éléments de la forêt et `reduce` permettant de concaténer ces paragraphes.

- Comme d'habitude, pour concaténer les paragraphes, nous appliquons `reduce` à la fonction `paragraphe-append` et au paragraphe vide.
- Pour fabriquer la liste des paragraphes, nous devons appliquer la fonction `findAux` à chacun des éléments de la forêt, mais avec un autre argument, le préfixe : nous définissons donc une fonction interne qui a ce dernier argument comme variable globale.

D'où la définition :

```
(define (find ident systeme)
  ;; findAux : string * Systeme -> Paragraphe
  ;; (findAux path systeme) rend le paragraphe dont les lignes sont constituées par les
  ;; noms complets des fichiers ou répertoires du système «systeme» dont le nom est
  ;; «ident», chaque ligne étant préfixée par «path».
  ... cf. ci-dessus
  ;; find-foret : string * Foret[Descripteur] -> Paragraphe
  ;; (find-foret path F) rend le paragraphe obtenu en concaténant, pour tout arbre «A»
  ;; de la fôret «F», (findAux path A)
  (define (find-foret path F)
    ; findAux-path : ArbreGeneral[Descripteur] -> Paragraphe
    ; (findAux-path G) rend (findAux path G)
    (define (findAux-path G)
      (findAux path G))
    (reduce paragraphe-append (paragraphe '()) (map findAux-path F)))
  ...
```

Programmation récursive

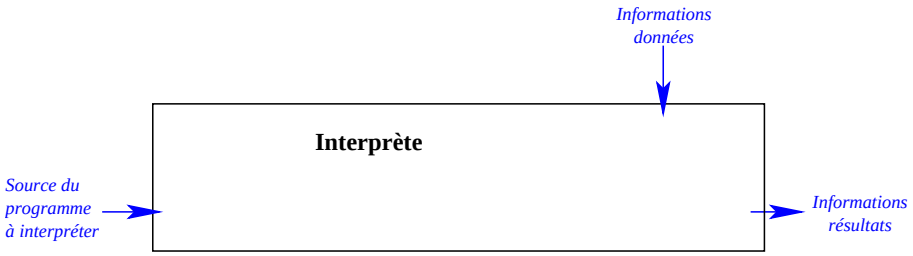
Troisième saison

Version 1.3

Sommaire

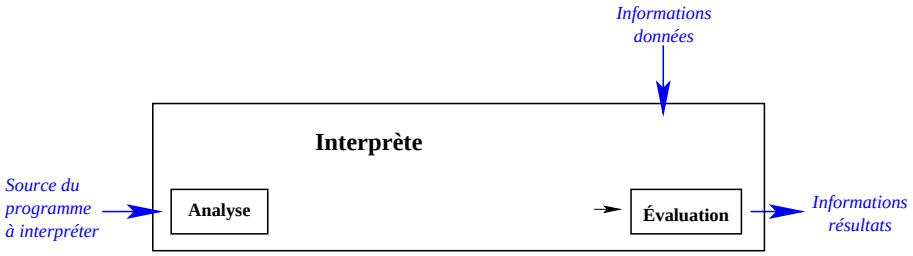
1. Introduction

Dans l'introduction de la première saison, nous avons dit qu'un interprète pouvait être schématisé par :

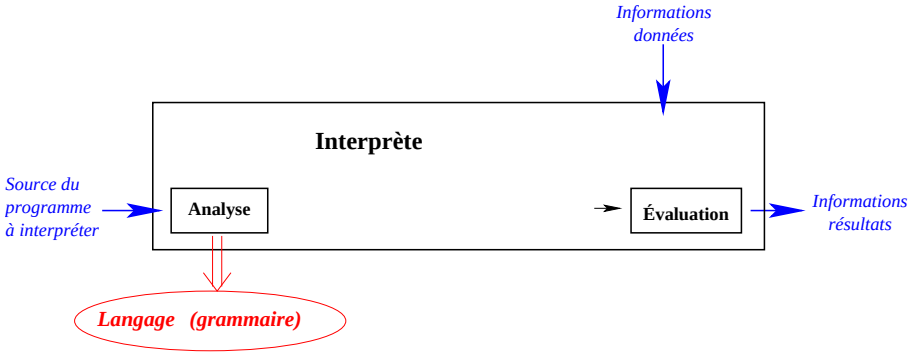


données, programmes (et résultats) étant des informations (sous forme de textes).
Remarque : dans ce cours, nous n'utilisons pratiquement pas la possibilité de lire des données (informations données).

Ainsi, un interprète doit « comprendre » le programme et les données et faire calculer par l'ordinateur le résultat en utilisant l'algorithme, ou le procédé de calcul, décrit par le programme. La « compréhension » s'effectue dans une phase dite d'*analyse*, le calcul s'effectuant dans une phase dite d'*évaluation* :

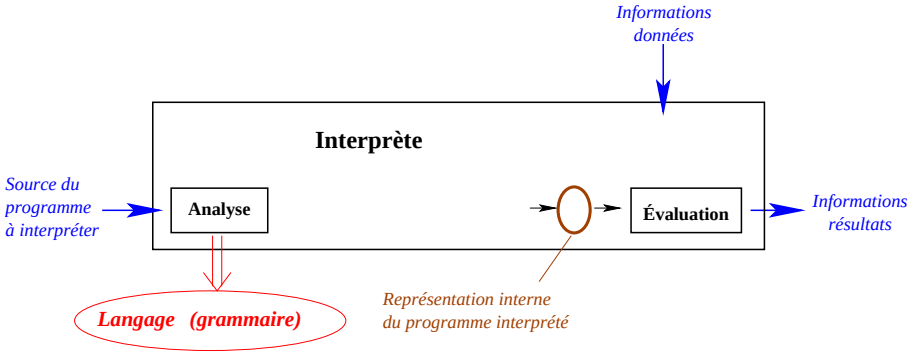


On peut analyser le source du programme parce qu'il est écrit dans un *langage* et on peut le faire analyser par un programme d'ordinateur parce que ce langage est parfaitement défi ni, en général à l'aide d'une grammaire :

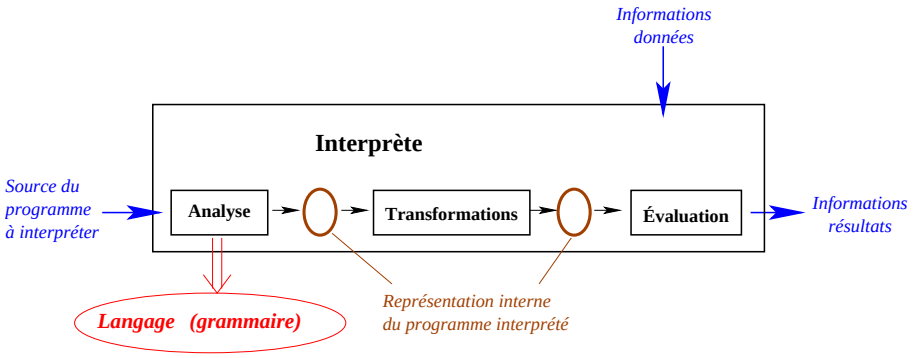


Noter que cette analyse peut être un problème diffi cile – de plus en plus lorsque l'on veut se rapprocher d'une langue naturelle – et qu'il existe des concepts et des techniques extrêmement sophistiqués pour le résoudre. Dans ce cours, nous n'analyserons que des langages simples (nous vous fournirons un analyseur pour un langage plus compliqué à analyser) mais nous aurons tout de même besoin de quelques concepts pour nous aider (nous étudierons ces concepts dans la première section).

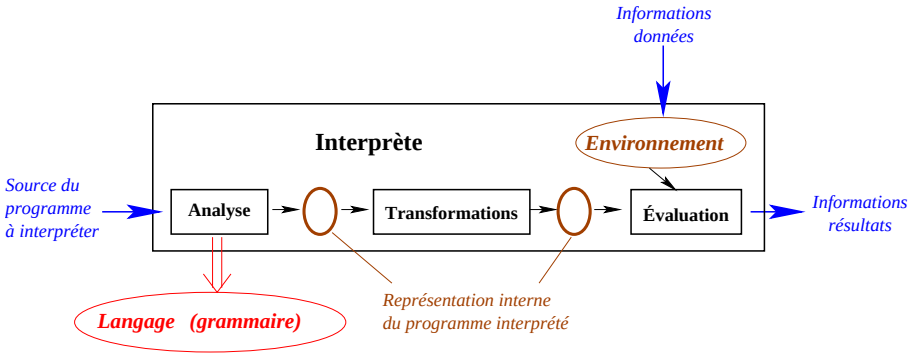
Afin que l'évaluation soit le plus efficace possible (et que le source de l'interprète correspondant soit plus simple à écrire), la phase d'analyse construit une représentation interne du programme à interpréter. Il faut bien voir que le choix de la structure de données contenant cette représentation interne est de la plus haute importance pour l'efficacité de l'interpréteur.



Pour améliorer l'efficacité de l'évaluation ou pour simplifier l'écriture de l'évaluateur (en diminuant le nombre de constructions à traiter), on peut aussi transformer la représentation du programme en une autre équivalente (c'est-à-dire dont l'évaluation donnera le même résultat) :



Enfin, dès qu'il y a la notion de variable dans le langage, l'évaluateur évalue le programme dans un *environnement*, structure de données qui, entre autres, reçoit les données du problème (noter que cet environnement évolue lors d'une évaluation) :



Dans les sections qui suivent, nous introduirons les concepts et les techniques en nous appuyant sur quatre exemples qui tournent autour des langages logiques.

2. Notion de grammaire

Avant de parler de grammaire, donnons le sens que nous utiliserons pour les mots « expression » et « langage ». Une **expression** est une représentation textuelle d'un calcul. L'ensemble des expressions utilisables est un **langage**.

2.1. Exemples de grammaires des langages des expressions booléennes simples

Comme premier exemple, étudions les expressions booléennes simples (*i.e.* la disjonction et la conjonction sont de opérateurs binaires), avec variables.

Comme nous l'avons rappelé dans le paragraphe **??**, page **??**, il existe plusieurs écritures d'une expression selon le placement de l'opérateur par rapport à ses opérandes (écriture infixe, préfixe...) et selon le status des parenthèses (écriture sans parenthèse, complètement parenthésée...). Nous donnons ci-dessous quatre définitions, à l'aide de grammaires, de langages des expressions booléennes simples.

2.1.1. En infixé, complètement parenthésée

Voici la grammaire des expressions booléennes simples avec variables en infixé complètement parenthésée :

$\langle expBoolSimple_{infixe} \rangle \rightarrow \langle atomique \rangle \mid \langle unaire \rangle \mid \langle binaire \rangle$

$\langle unaire \rangle \rightarrow$ ($\langle opérateur1 \rangle$ $\langle expBoolSimple_{infixe} \rangle$)

$\langle binaire \rangle \rightarrow$ ($\langle expBoolSimple_{infixe} \rangle$ $\langle opérateur2 \rangle$ $\langle expBoolSimple_{infixe} \rangle$)

$\langle atomique \rangle \rightarrow \langle constante \rangle \mid \langle variable \rangle$

$\langle constante \rangle \rightarrow$ `@f` FAUX | `@v` VRAI

$\langle variable \rangle \rightarrow$ *Une suite de caractères autres que l'espace, les parenthèses et @*

$\langle opérateur1 \rangle \rightarrow$ `@non`

$\langle opérateur2 \rangle \rightarrow$ `@et` ET | `@ou` OU

Terminologie :

- $\langle expBoolSimple_{infixe} \rangle$, $\langle constante \rangle$... sont des **non-terminaux** ou **unités syntaxiques** de la grammaire ; ils ne seront jamais écrits tels quels, on les dérivera (cf. plus loin) ;
- `@f` , `@v` , `@non` , `@et` et `@ou` sont des éléments **terminaux** de la grammaire ; ils sont écrits tels quels dans le langage ;
- *Une suite de caractères autres que l'espace, les parenthèses et @* est aussi un élément terminal mais, contrairement aux précédents, on le décrit de façon informelle ;
- $\langle expBoolSimple_{infixe} \rangle$, qui est la première unité syntaxique définie, est l'axiome de la grammaire : c'est d'elle dont on part pour définir un mot du langage.

Remarque : nous voulons écrire des fonctions Scheme manipulant des mots du langage, aussi avons-nous dû choisir des graphismes utilisables sur ordinateur : les opérateurs booléens classiques, $\wedge$, $\vee$ et $\neg$, ont été remplacés par `@et` , `@ou` et `@non` . D'autre part, pour bien distinguer les variables (que l'on a tendance à noter *v*) des constantes booléennes, nous avons noté ces dernières `@f` et `@v` . Noter que nous n'avons pas pris `#f` et `#t` car nous voulons bien distinguer le langage Scheme et notre langage des expressions booléennes.

2.1.2. « À la Scheme »

On peut aussi décider d'écrire les expressions « à la Scheme » (*i.e.* en préfixé, complètement parenthésées) :

$\langle expBoolSimple_{Scheme} \rangle \rightarrow \langle atomique \rangle \mid \langle unaire \rangle \mid \langle binaire \rangle$

$\langle unaire \rangle \rightarrow$ ($\langle opérateur1 \rangle$ $\langle expBoolSimple_{Scheme} \rangle$)

$\langle binaire \rangle \rightarrow$ ($\langle opérateur2 \rangle$ $\langle expBoolSimple_{Scheme} \rangle$ $\langle expBoolSimple_{Scheme} \rangle$)

$\langle atomique \rangle \rightarrow \langle constante \rangle \mid \langle variable \rangle$

<constante> → @f FAUX
@v VRAI

<variable> → *Une suite de caractères autres que l'espace, les parenthèses et @*

<opérateur1> → @non

<opérateur2> → @et ET
@ou OU

2.1.3. En polonaise préfixée

On peut aussi décider d'écrire les expressions en polonaise préfi xée (sans parenthèse) :

<expBoolSimple$_{polonaise}$> → *<atomique>*
<unaire>
<binaire>

<unaire> → *<opérateur1> <expBoolSimple$_{polonaise}$>*

<binaire> → *<opérateur2> <expBoolSimple$_{polonaise}$> <expBoolSimple$_{polonaise}$>*

<atomique> → *<constante>*
<variable>

<constante> → @f FAUX
@v VRAI

<variable> → *Une suite de caractères autres que l'espace, les parenthèses et @*

<opérateur1> → @non

<opérateur2> → @et ET
@ou OU

2.1.4. En notation « normale »

<expBoolSimple$_{normale}$> → *<atomique>*
<unaire>
<binaire>
(*<expBoolSimple$_{normale}$>*)

<unaire> → *<opérateur1> <expBoolSimple$_{normale}$>*

<binaire> → *<expBoolSimple$_{normale}$> <opérateur2> <expBoolSimple$_{normale}$>*

<atomique> → *<constante>*
<variable>

<constante> → @f FAUX
@v VRAI

<variable> → *Une suite de caractères autres que l'espace, les parenthèses et @*

<opérateur1> → @non

<opérateur2> → @et ET
@ou OU

Exemple 1 : `@non (@v @et @v) @ou @f`
Exemple 2 : `@non @v @et (@v @ou @f)`
Exemple 3 : `@non @v @et @v @ou @f`
Remarques :

- Pour savoir comment calculer l'expression `@non @v @et @v @ou @f`, il faut savoir qu'elle est équivalente à `((@non @v) @et @v) @ou @f`. Pour ce faire, on doit, en plus de la grammaire, donner des règles de priorités : ici, pour suivre les conventions classiques, nous décidons que le « `@non` » est plus prioritaire que le « `@et` » lui-même plus prioritaire que le « `@ou` ». Ces règles de priorités imposent de parenthéser d'abord les sous-expressions correspondant à un « `@non` » et, ensuite, celles qui correspondent à un « `@et` » (et on n'a pas besoin de parenthéser les sous-expressions qui correspondent à « `@ou` » car c'est la priorité la plus faible). Ainsi, dans l'exemple `@non @v @et @v @ou @f`,
  - il y a un « `@non` » (opérateur unaire) : cette expression est équivalente à `(@non @v) @et @v @ou @f`,

– il y a un « @et » (opérateur binaire) : cette expression est équivalente à ((@non @v) @et @v) @ou @f.
– La règle

  <expBoolSimple$_{normale}$> → (*<expBoolSimple$_{normale}$>*)

  permet alors de parenthéser des sous-expressions qui, sinon, seraient découpées autrement par les règles de priorités (par exemple `@non (@v @et @v) @ou @f`.

Notes didactiques :

– ce langage est étudié parce que ce sont les expressions booléennes que vous avez l'habitude d'utiliser par ailleurs ;
– la description du langage que nous avons donné ci-dessus utilise autre chose (la notion de priorité) que la notion de grammaire ;
– il existe des grammaires auto-suffisantes pour ce langage mais elles s'éloignent de la forme des grammaires que nous avons données pour décrire les autres langages des expressions booléennes simples et, surtout, elles demandent une réflexion approfondie sur les grammaires, étude qui dépasse le cadre de cet ouvrage.

2.1.5. Similitude des différents ensembles de règles de grammaire

On peut noter que toutes ces grammaires ont des points en commun, points en commun que nousretrouverions dans tous les langages des expressions booléennes simples :

– les règles pour les expressions atomiques sont exactement les mêmes,
– surtout, les règles pour les expressions binaires disent que l'on doit décomposer l'expression avec :
  – un opérateur (qui peut être à différentes places selon le langage),
  – qui a deux opérandes qui sont des expressions (que l'on devra donc aussi décomposer),
– de même, les règles pour les expressions unaires disent toutes que l'on doit décomposer l'expression avec :
  – un opérateur (qui peut être à différentes places selon le langage),
  – qui a un opérande qui est une expression (que l'on devra donc aussi décomposer).

On peut donc parler d'une classe de langages définis par une classe de grammaires (ou par une classe de langages définie par une classe de grammaires).

2.2. Génération d'un mot du langage

Considérons un langage de la classe des langages des expressions booléennes simple, par exemple celui des expressions écrites en infixé, complètement parenthésé.

Le langage défini par une grammaire est égal à l'ensemble des mots (ou, si vous préférez, expressions) que l'on peut générer à l'aide de la grammaire.

Notation : dans les schémas de ce paragraphe, pour des raisons de mise en page, les unités syntaxiques *<expBoolSimple$_{infixe}$>*, *<atomique>*, *<constante>*, *<opérateur1>* et *<opérateur2>* sont renommées respectivement *<expr>*, *<atom>*, *<cste>*, *<op1>* et *<op2>*.

Pour générer un mot du langage à partir de la grammaire, on part de l'axiome (rappelons que c'est la première unité syntaxique définie dans la grammaire), ici *<expBoolSimple$_{infixe}$>* que nous avons renommé *<expr>* :

expr

et on la dérive, c'est-à-dire que l'on prend une des possibilités de la règle et qu'on remplace l'axiome par le membre droit de cette possibilité. Par exemple, en prenant la troisième possibilité :

binaire

<binaire> est un non-terminal : on doit effectuer la dérivation (il n'y a qu'une possibilité) :

(*expr* *op2* *expr*)

les deux parenthèses sont des terminaux : on ne les touche pas ; en revanche, on dérive comme ci-dessus les non-terminaux *<expr>*, *<op2>* et *<expr>* (remarquez que l'axiome joue le même rôle que n'importe quel non-terminal, sauf au départ). Prenons la deuxième possibilité pour le premier *<expr>*, la seconde possibilité pour *<op2>* et la première possibilité pour le second *<expr>* :

unaire *@ou* *atom*

@et est un terminal : on ne le touche pas ; dérivons les terminaux *<unaire>* (il n'y a qu'une posibilité) et *<atom>* (qui correspond à *<atomique>*) en prenant la première possibilité :

pour <*op1*>, il n'y a qu'une possibilité ; pour <*expr*>, décidons de prendre la troisième possibilité ; pour <*cste*> (qui correspond à <*constante*>), prenons la première possibilité :

$$\overbrace{@non}\quad\overbrace{\text{binaire}}\qquad\overbrace{@f}$$

le seul non-terminal est <*binaire*> et il n'y a qu'une possibilité :

$$\overbrace{(\quad \text{expr}\quad \text{op2}\quad \text{expr}\quad)}$$

décidons de prendre la première possibilité pour les deux occurrences de <*expr*> et la première possibilité pour <*op2*> :

$$\overbrace{\text{atom}}\quad\overbrace{@et}\quad\overbrace{\text{atom}}$$

décidons de prendre la première possibilité pour les deux occurrences de <*atom*> :

$$\overbrace{\text{cste}}\qquad\overbrace{\text{cste}}$$

et enfi n dérivons les deux occurrences du non-terminal <*cste*> en prenant la seconde possibilité dans les deux cas :

$$\overbrace{@v}\qquad\overbrace{@v}$$

Toutes les dérivations ont été effectuées (il ne reste plus de non-terminaux en suspens). Pour connaître le mot généré, il suffi t de lire de gauche à droite les terminaux :

((@non (@v @et @v)) @ou @f)

2.2.1. Constructeurs

Par programme, on ne génère pas l'ensemble des mots du langage, mais on construit souvent des mots du langage (par exemple, dans la suite de cet ouvrage, on simplifi era des expressions : il s'agit de construire une expression à partir d'une expression donnée).

Pour ce faire, on doit pouvoir effectuer exactement ce que nous avons fait « à la main » lorsque nous avons généré un mot à partir de la grammaire. Autrement dit, nous devons pouvoir écrire des constantes (vrai et faux), des variables, des opérateurs (non, et, ou) et des expressions composées (unaires et binaires), ce qui sera fait en utilisant des **construc-teurs** (`ebs-constante-vrai`, `ebs-constante-faux`, `ebs-variable`, `ebs-unaire`, `ebs-binaire`, `ebs-operateur1-non`, `ebs-operateur2-et` et `ebs-operateur2-ou`).

Règles de nomenclature : pour faciliter la lecture des programmes (et donc leur écriture), il est important de se donner, et de suivre, des règles pour le choix des noms des fonctions. Nous préciserons plus loin les différentes règles que nous suivons, mais notons tout de suite que, systématiquement, les noms des fonctions utiles pour un langage seront préfi xés par le nom abrégé du langage (ici « ebs- » pour « expressions booléennes simples »).

2.3. Analyse d'un mot

Les grammaires sont utilisées, particulièrement en informatique, pour savoir si un mot (une phrase) appartient bien à un certain langage (penser aux langages de programmation, aux langages de commande...) et pour effectuer une certaine tâche lorsque c'est le cas. On nomme **analyse d'un mot** la décomposition de ce mot, décomposition effectuée pour savoir si le mot appartient au langage ou pour déterminer la tâche qu'il exprime.

Dans ce paragraphe, nous voudrions voir comment écrire un programme qui réalise cette analyse. Pour ce faire, regardons comment on pourrait analyser « à la main » un mot du langage, en dégageant les fonctions qui seront nécessaires pour écrire un tel programme d'analyse.

2.3.1. Reconnaisseurs

Supposons que l'on veuille savoir si

$$((@non\ (@v\ @et\ @v))\ @ou\ @f)$$

est un mot du langage engendré par la grammaire précédente. Pour que ce soit le cas, il faut qu'il dérive de l'axiome <*expBoolSimple*$_{infixe}$> :

?? <*expBoolSimple*$_{infixe}$> $\Rightarrow$ $((@non\ (@v\ @et\ @v))\ @ou\ @f)$??

La règle de <*expBoolSimple*$_{infixe}$> fournit trois possibilités. Laquelle choisir ? Ici,

– on voit facilement que le mot ne peut pas dériver du non-terminal *<atomique>* (car les mots qui dérivent de ce non terminal ne commencent pas par une parenthèse ouvrante),
– on peut voir qu'il ne peut pas dériver du non-terminal *<unaire>* (car le second caractère d'un tel mot est toujours un « @ »),
– on peut voir qu'il est possible qu'il dérive du non-terminal *<binaire>*.

Si l'on veut écrire un programme qui analyse un mot afin de déterminer s'il appartient au langage engendré par la grammaire, il faut avoir des fonctions pour décider dans quel cas on est. On nomme ces fonctions – qui sont des prédicats – des **reconnaisseurs**.

Par exemple, avec notre grammaire des expressions booléennes simples, nous devons avoir les reconnaisseurs `ebs-atomique?`, `ebs-unaire?` et `ebs-binaire?`.

Attention, en général, on ne peut pas demander que les reconnaisseurs soient infaillibles, tout en exigeant que le programme d'analyse, lui, le soit. Pour s'en convaincre il suffit de se dire que la définition de la fonction d'analyse serait alors triviale (or cela se saurait si la définition des fonctions d'analyse était triviale : c'est un problème qui a occupé de nombreux chercheurs dans les années 50 et 60) :

```
(define (ebs-infixe? m)
  (or (ebs-atomique? m) (ebs-unaire? m) (ebs-binaire? m)))
```

Ce qu'on demande aux reconnaisseurs, c'est qu'ils nous aiguillent vers la bonne règle lorsque le mot appartient au langage (il serait peut-être plus judicieux de les appeler « aiguilleurs » mais classiquement on les nomme reconnaisseurs). Ainsi, en général, les corps des définitions de fonctions dont la donnée est `exp`, supposé du langage, seront de la forme :

```
(cond ((ebs-atomique? exp)
       ; expression de la valeur à rendre lorsque exp dérive de <atomique>
       )
      ((ebs-unaire? exp)
       ; expression de la valeur à rendre lorsque exp dérive de <unaire>
       )
      ((ebs-binaire? exp)
       ; expression de la valeur à rendre lorsque exp dérive de <binaire>
       )
      (else (erreur 'fn "donnée pas bien formée")))
```

2.3.2. Accesseurs

Revenons à l'analyse, « à la main », du mot $((@non\ (@v\ @et\ @v))\ @ou\ @f)$. Nous avons dit que ce mot ne pouvait dériver que du non-terminal *<binaire>*. Regardons si c'est possible :

$??\ <binaire> \Rightarrow ((@non\ (@v\ @et\ @v))\ @ou\ @f)\ ??$

La règle de *<binaire>* ne comporte qu'une possibilité : il n'y a donc pas de problème de choix. Le mot et la partie droite commencent par une parenthèse ouvrante (cela correspond donc) et elles finissent par une parenthèse fermante (cela correspond toujours) mais, dans la partie droite de la règle, il reste $<expBoolSimple_{infixe}>$ *<opérateur2>* $<expBoolSimple_{infixe}>$ qu'il faut faire correspondre avec $(@non\ (@v\ @et\ @v))\ @ou\ @f$.

À la main, on voit que la décomposition doit être $(@non\ (@v\ @et\ @v))$ pour la première occurrence de $<expBoolSimple_{infixe}>$, *@ou* pour *<opérateur2>* et $@f$ pour la seconde occurrence de $<expBoolSimple_{infixe}>$.

Si l'on veut écrire un programme il faut que l'on ait des fonctions pour accéder aux différents éléments de cette décomposition. On nomme **accesseurs** de telles fonctions. Ainsi, dans notre exemple, nous avons besoin des accesseurs `ebs-binaire-operande-gauche`, `ebs-binaire-operande-droit` et `ebs-binaire-operateur`.

Et on continue en analysant $(@non\ (@v\ @et\ @v))$ (qui doit dériver de $<expBoolSimple_{infixe}>$), *@ou* (qui doit dériver de *<opérateur2>*) et $@f$ (qui doit dériver de $<expBoolSimple_{infixe}>$) :

$??\ <expBoolSimple_{infixe}> \Rightarrow (@non\ (@v\ @et\ @v))\ ??$
...
$??\ <opérateur2> \Rightarrow @ou\ ??$
...
$??\ <expBoolSimple_{infixe}> \Rightarrow @f\ ??$
...

3. Notion de barrière syntaxique

Nous avons vu que les différentes grammaires des expressions booléennes simples définissaient une classe de langages. Aussi les spécifications des reconnaisseurs, des accesseurs et des constructeurs seront exactement les mêmes dans tous les cas (si vous ne nous croyez pas – ce qui serait plus que légitime –, redéfinissez-les pour les expressions préfixes).

D'autre part, nous avons vu que pour analyser un mot d'un langage des expressions booléennes simples, il fallait avoir

- les reconnaisseurs `ebs-atomique?`, `ebs-variable?`, `ebs-constante?`, `ebs-constante-vrai?`, `ebs-constante-faux?`, `ebs-unaire?`, `ebs-binaire?`, `ebs-operateur1?`, `ebs-non?`, `ebs-operateur2?`, `ebs-operateur2-et?`, `ebs-operateur2-ou?`,
- les accesseurs
  - `ebs-unaire-operande`, `ebs-unaire-operateur`,
  - `ebs-binaire-operande-gauche`, `ebs-binaire-operande-droit` et `ebs-binaire-operateur`.

et que pour construire des expressions booléennes simples, il fallait avoir

- les constructeurs `ebs-variable`, `ebs-constante-vrai`, `ebs-constante-faux`, `ebs-unaire`, `ebs-binaire`, `ebs-operateur1-non`, `ebs-operateur2-et` et `ebs-operateur2-ou`.

Il est clair que l'implantation de ces différentes fonctions est interdépendante et dépend du choix que l'on fait pour implanter les expressions. On doit donc regrouper cet ensemble de fonctions dans une barrière d'abstraction, barrière d'abstraction pour une classe de grammaires, qu'on appelle la **barrière syntaxique**.

Nous allons donner la spécification précise de ces fonctions après quelques généralités.

3.1. Schéma des spécifications des fonctions ayant comme donnée un mot du langage

Systématiquement, nous nommerons le type des mots qui dérivent d'un non-terminal par le nom de celui-ci (sans accents), avec la première lettre en majuscule. Par exemple, le type des expressions représentés par des mots dérivant de $<expBoolSimple_{infixe}>$ sera noté `ExpBoolSimple`, le type des mots dérivant de <*unaire*> sera noté `Unaire` ..., et la spécification des fonctions qui ont comme données un tel type est du genre :

```
;;; fn1: ExpBoolSimple -> ...
;;; (fn1 exp) rend ...

;;; fn2: Unaire -> ...
;;; (fn2 exp) rend ...
```

Rappelons qu'une telle spécification signifie que l'on garantit que la fonction rend bien ce qui est dit après « rend » lorsque la donnée appartient au type donné (`ExpBoolSimple` dans le cas de `fn1`, `Unaire` dans le cas de `fn2`) et que l'on ne sait pas ce qu'elle fait (signifie une erreur ou rend une valeur plus ou moins significative) dans le cas contraire. On pourra donner des précisions dans la spécification :

- si nous savons qu'une erreur est signifiée sous certaines conditions, nous l'indiquerons dans la spécification comme d'habitude :

```
;;; fn3: ExpBoolSimple -> ...
;;; ERREUR lorsque ...
;;; (fn3 exp) rend ...
```

- pour de nombreuses fonctions, nous saurons quelle valeur elles rendent lorsque la donnée n'appartient pas au type ; nous l'indiquerons alors explicitement dans la spécification :

```
;;; fn4: ExpBoolSimple -> ...
;;; (fn4 exp) rend ...
;;; (elle rend ... lorsque ...)
```

(nous verrons des exemples concrets ci-après).

Remarque : ces précisions sont particulièrement utiles pour les reconnaisseurs car elles permettent de donner des messages d'erreur plus significatifs lorsque le mot à analyser n'appartient pas au langage.

3.2. Spécifications des reconnaisseurs

Noter que lorsqu'on applique les reconnaisseurs `ebs-atomique?` , `ebs-unaire?` et `ebs-binaire?` , on espère que le mot à analyser dérive du non-terminal *<expBoolSimple$_{infixe}$>*. Le type de la donnée de ces reconnaisseurs est donc `ExpBoolSimple` (et le type d'arrivée est bien sûr `bool`).

Pour `ebs-atomique?` , nous avons dit qu'il était aisé de savoir si un mot quelconque pouvait dériver ou non de *<atomique>*. On peut donc prendre comme spécifi cation :

```
;;; ebs-atomique? : ExprBoolSimple -> bool
;;; (ebs-atomique? exp) rend #t ssi exp est une expression atomique
;;; (constante ou variable)
```

Pour `ebs-unaire?` , nous avons dit qu'il était diffi cile, voire impossible, de savoir si un mot quelconque pouvait dériver ou non de *<unaire>*. En revanche, il peut être aisé de savoir si on peut bien le décomposer en deux composants. D'où la spécifi cation :

```
;;; ebs-unaire? : ExprBoolSimple -> bool
;;; (ebs-unaire? exp) rend #t ssi exp ne peut être qu'une expression dérivant
;;; de <unaire>
;;; (elle rend #f lorsque exp n'est pas composée de deux composants)
```

De même, pour `ebs-binaire?` , il est aisé de savoir si on peut bien le décomposer en trois composants. D'où la spécifi cation :

```
;;; ebs-binaire? : ExprBoolSimple -> bool
;;; (ebs-binaire? exp) rend #t ssi exp ne peut être qu'une expression
;;; dérivant de <binaire>
;;; (elle rend #f lorsque exp n'est pas composée de trois composants)
```

Lorsqu'on applique les reconnaisseurs `ebs-variable?` et `ebs-constante?` , on espère que le mot à analyser dérive du non-terminal *<atomique>*. Le type de la donnée de ces reconnaisseurs est donc `Atomique` :

```
;;; ebs-variable? : Atomique -> bool
;;; (ebs-variable? exp) rend #t ssi exp est une variable
```

```
;;; ebs-constante? : Atomique -> bool
;;; (ebs-constante? exp) rend #t ssi exp est une constante
;;; (rend #f lorsque exp n'est pas une expression atomique)
```

Noter que la spécifi cation de la fonction `ebs-constante?` que nous avons donnée indique qu'en fait ce reconnaisseur rend toujours (y compris lorsque l'on ne sait pas que le mot doit dériver de *<atomique>*) la bonne réponse.

Donnons les spécifi cations des autres reconnaisseurs sans autres commentaires :

```
;;; ebs-constante-vrai? : Constante -> bool
;;; (ebs-constante-vrai? exp) rend #t ssi exp est la constante @V
;;;   (rend #f lorsque exp n'est pas une constante)
```

```
;;; ebs-constante-faux? : Constante -> bool
;;; (ebs-constante-faux? exp) rend #t ssi exp est la constante @F
;;;   (rend #f lorsque exp n'est pas une constante)
```

```
;;; ebs-operateur1? : Symbole -> bool
;;; (ebs-operateur1? s) rend #t ssi s est un opérateur unaire
```

```
;;; ebs-operateur2? : Symbole -> bool
```

;;; *(ebs-operateur2? s) rend #t ssi s est un opérateur binaire*

;;; *ebs-operateur1-non? : Symbole -> bool*
;;; *(ebs-operateur1-non? s) rend #t ssi s est l'opérteur (unaire) non*

;;; *ebs-operateur2-et? : Symbole -> bool*
;;; *(ebs-operateur2-et? s) rend #t ssi s est l'opérteur (binaire) et*

;;; *ebs-operateur2-ou? : Symbole -> bool*
;;; *(ebs-operateur2-ou? s) rend #t ssi s est l'opérteur (binaire) ou*

3.3. Spécifications des accesseurs

Noter que lorqu'on applique les accesseurs `ebs-unaire-operande` et `ebs-unaire-operateur`, on espère que le mot à analyser dérive du non-terminal <*unaire*>. Le type de la donnée de ces reconnaisseurs est donc `Unaire`. Le type d'arrivée est le type qui correspond au non-terminal extrait. Par exemple, la règle étant

<*unaire*> → (<*opérateur1*> <*expBoolSimple*$_{infixe}$>)

`ebs-unaire-operateur` doit rendre un élément de `Operateur1` et `ebs-unaire-operande` doit rendre un élément de `expression`. Noter aussi que l'on ne peut rien dire lorsque le mot donné ne dérive pas de <*unaire*>. Les spécifications sont donc :

;;; *ebs-unaire-operateur : Unaire -> Operateur1*
;;; *(ebs-unaire-operateur exp) rend l'opérateur (principal) de l'expression*
;;; *donnée*

;;; *ebs-unaire-operande : Unaire -> ExprBoolSimple*
;;; *(ebs-unaire-operande) rend l'expression, opérande de l'expression donnée*

De même, les spécifications des accesseurs correspondant à la règle

<*binaire*> → (<*expBoolSimple*$_{infixe}$> <*opérateur2*> <*expBoolSimple*$_{infixe}$>)

sont :

;;; *ebs-binaire-operande-gauche : Binaire -> ExprBoolSimple*
;;; *(ebs-binaire-operande-gauche exp) rend l'expression, opérande gauche de*
;;; *l'expression donnée*

;;; *ebs-binaire-operande-droit : Binaire -> ExprBoolSimple*
;;; *(ebs-binaire-operande-droit exp) rend l'expression, opérande droit de*
;;; *l'expression donnée*

;;; *ebs-binaire-operateur : Binaire -> Operateur2*
;;; *(ebs-binaire-operateur exp) rend l'opérateur (principal) de l'expression*
;;; *donnée*

Remarque : noter la règle de nommage des fonctions que nous utilisons (un préfixe correspondant au non-terminal analysé, un suffixe correspondant à l'unité syntaxique extraite).

3.4. Spécifications des constructeurs

```
;;;; Constructeurs:

;;; ebs-variable : Symbole -> ExprBoolSimple
;;; (ebs-variable s) rend l'expression booléenne simple réduite à la variable s


;;; ebs-constante-vrai : -> Constante
;;; (ebs-constante-vrai) rend la constante « vrai »


;;; ebs-constante-faux : -> Constante
;;; (ebs-constante-faux) rend la constante « faux »


;;; ebs-unaire : Operateur1 * ExprBoolSimple -> ExprBoolSimple
;;; (ebs-unaire op exp) rend l'expression composée unaire ayant comme
;;; opérateur op et comme opérande exp


;;; ebs-binaire : ExprBoolSimple * Operateur2 * ExprBoolSimple -> ExprBoolSimple
;;; (ebs-binaire exp1 op exp2) rend l'expression composée ayant comme
;;; opérande gauche exp1, comme opérateur op et comme opérande droit
;;; exp2


;;; ebs-operateur1-non : -> Operateur1
;;; (ebs-operateur1-non) rend l'opérateur de négation


;;; ebs-operateur2-et : -> Operateur2
;;; (ebs-operateur2-et) rend l'opérateur de conjonction


;;; ebs-operateur2-ou : -> Operateur2
;;; (ebs-operateur2-ou) rend l'opérateur de disjonction
```

Exemple : pour construire l'expression booléenne, représentée en notation infi xée, complètement parenthésée, par :

((@non (@v @et @v)) @et (@v @ou @f))

on peut écrire :

```
(ebs-binaire    (ebs-unaire    (ebs-operateur1-non)
                               (ebs-binaire    (ebs-constante-vrai)
                                               (ebs-operateur2-et)
                                               (ebs-constante-vrai)))
                (ebs-operateur2-et)
                (ebs-binaire    (ebs-constante-vrai)
                                (ebs-operateur2-ou)
                                (ebs-constante-faux)))
```

3.5. Premier exemple d'utilisation de la barrière syntaxique

Écrivons la défi nition d'une fonction qui teste si une expression booléenne simple est une expression constante (*i.e.* qui ne comporte pas de variable).

Spécification

La spécifi cation de cette fonction est :

```
;;; ebs-exp-cste? : ExprBoolSimple -> bool
```

;;; *(ebs-exp-cste? exp) rend #t ssi exp est une expression constante*
;;; *(c'est-à-dire qui n'a pas d'occurrence de variable)*

Exemple d'application

```
(let ((expression
        (ebs-binaire    (ebs-unaire    (ebs-operateur1-non)
                                       (ebs-constante-faux))
                        (ebs-operateur2-et)
                        (ebs-binaire    (ebs-variable    'a)
                                        (ebs-operateur2-ou)
                                        (ebs-variable    'b)))))
  (ebs-exp-cste?    expression))    → #f
```

Implantation

Comme pour toutes les fonctions dont la donnée est un mot du langage, la forme générale de la définition de ebs-exp-cste ? est une conditionnelle qui passe en revue toutes les branches de la règle donnée pour l'axiome (rappelons que c'est la première règle) :

```
(define  (ebs-exp-cste?    exp)
  (cond
    ((ebs-atomique?    exp)
     ... à faire
    ((ebs-unaire?    exp)
     ... à faire
    ((ebs-binaire?    exp)
     ... à faire
     ... à faire
    (else  (erreur  'ebs-exp-cste?
                    "expression    mal  formée"))))
```

et il n'y a plus qu'à remplir les trous. Lorsque l'expression est une expression atomique, elle est constante si, et seulement si, le reconnaisseur ebs-constante? rend #t :

```
(define  (ebs-exp-cste?    exp)
  (cond
    ((ebs-atomique?    exp)
     (ebs-constante?    exp))
    ((ebs-unaire?    exp)
     ... à faire
    ((ebs-binaire?    exp)
     ... à faire
     ... à faire
    (else  (erreur  'ebs-exp-cste?
                    "expression    mal  formée"))))
```

lorsque l'expression donnée est une expression unaire, elle est constante si, et seulement si, son opérande est une expression constante :

```
(define  (ebs-exp-cste?    exp)
  (cond
    ((ebs-atomique?    exp)
     (ebs-constante?    exp))
    ((ebs-unaire?    exp)
     (ebs-exp-cste?    (ebs-unaire-operande    exp)))
    ((ebs-binaire?    exp)


    (else  (erreur  'ebs-exp-cste?
                    "expression    mal  formée"))))
```

lorsque l'expression donnée est une expression binaire, elle est constante si, et seulement si, ses deux opérandes sont

des expressions constantes :

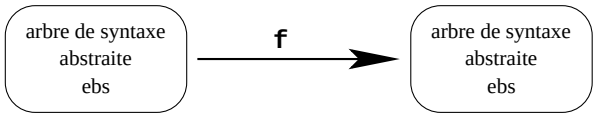
```
(define   (ebs-exp-cste?     exp)
  (cond
    ((ebs-atomique?     exp)
     (ebs-constante?     exp))
    ((ebs-unaire?     exp)
     (ebs-exp-cste?     (ebs-unaire-operande       exp)))
    ((ebs-binaire?     exp)
     (and  (ebs-exp-cste?     (ebs-binaire-operande-gauche         exp))
           (ebs-exp-cste?     (ebs-binaire-operande-droit         exp))))
    (else  (erreur   'ebs-exp-cste?
                     "expression    mal  formée"))))
```

4. Fonctions de lecture et d'écriture

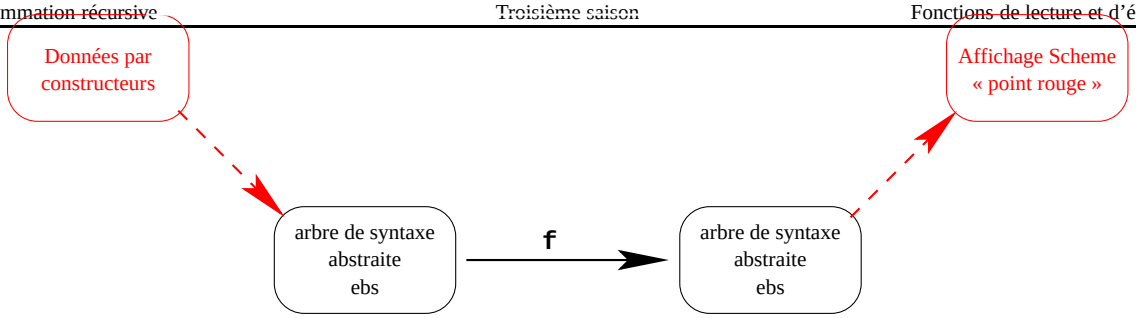
Une fonction manipulant les expressions booléennes simples peut avoir comme donnée ou comme résultat une expression booléenne simple :

- Lorsque la fonction rend une expression booléenne simple, au moins pour la tester, il faut pouvoir afficher une telle expression (rappelons que dans l'implantation que nous vous fournissons, toutes les expressions sont affichées avec un point rouge).
- Lorsque la fonction a comme donnée une expression booléenne simple, jusqu'à présent, pour la tester, nous avons « fabriqué » une telle expression booléenne en utilisant les constructeurs de la barrière syntaxique. Or, l'intérêt d'avoir un langage est d'utiliser des mots du langage comme données à de telles fonctions.

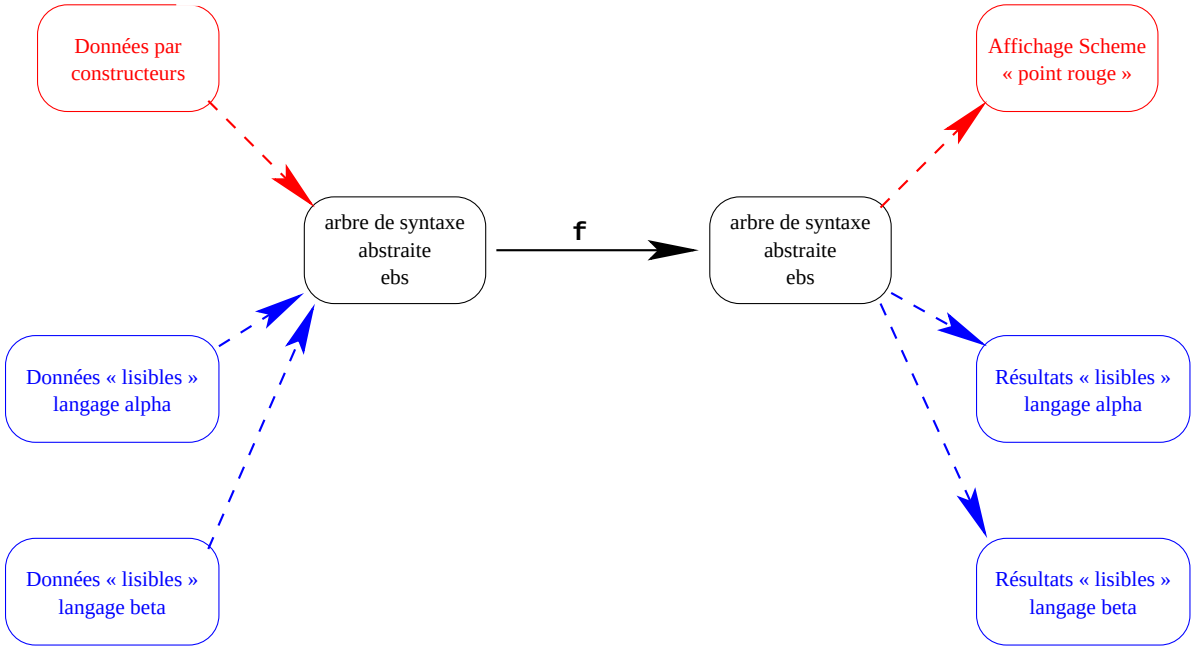
Considérons le cas d'une fonction, `f`, qui regroupe tous les problèmes, à savoir que la donnée et le résultat de la fonction sont des expressions booléennes simples :



Pour tester cette fonction, actuellement, travaillant uniquement avec la barrière syntaxique, la donnée doit être fournie uniquement à l'aide des constructeurs et l'affichage est, systématiquement, un point rouge (il faut être un expert plus que confirmé pour savoir si c'est le bon résultat...) :



Le but de cette section est d'écrire différentes fonctions de conversion pour pouvoir
- dans les programmes, écrire les données dans un langage lisible par l'homme (et comme les données sont des expressions booléennes simples, nous utiliserons des langages des expressions booléennes simples),
- convertir, pour que ce soit lisible par l'homme, des expressions booléennes simples dans un des langages des expressions booléennes simples :



Dans un premier temps, nous regardons comment convertir une expression booléenne simple en une expression lisible par l'homme (c'est facile, nous aurions pu le laisser en exercice). Dans un deuxième temps, nous regarderons la conversion d'une donnée lisible par l'homme en une expression booléenne simple.

4.2. Fonctions de conversion de sortie

La conversion est particulièrement simple si on se contente d'une forme complètement parenthésée car on a alors une Sexpression, structure que Scheme affiche parfaitement. Comme exercice, vous pouvez écrire les différentes fonctions (en préfixé, infixé et suffixé). Nous donnons ci-dessous une définition de la fonction qui convertit l'expression en préfixé (complètement parenthésée).

4.1.1. Conversion en préfixé (complètement parenthésée)

Nous nommons cette fonction `ebs->sexpr-pref` car sa donnée est une expression booléenne simple (d'où `ebs->`) et que son résultat est une Sexpression (d'où `sexpr`) représentant l'expression booléenne simple en préfixé (d'où `pref`).

Spécification

Sa spécification est :

```
;;; ebs->sexpr-pref : ExprBoolSimple -> Sexpression
;;; (ebs->sexpr-pref exp) rend la Sexpression représentant exp en
;;; notation préfixée, complètement parenthésée.
```

Exemple d'application

```
(let ((expression
       (ebs-binaire (ebs-unaire (ebs-operateur1-non)
                                (ebs-binaire (ebs-constante-vrai)
                                             (ebs-operateur2-et)
                                             (ebs-constante-vrai)))
                    (ebs-operateur2-et)
                    (ebs-binaire (ebs-constante-vrai)
                                 (ebs-operateur2-ou)
                                 (ebs-constante-faux)))))
  (ebs->sexpr-pref expression))
 → (@et (@non (@et @v @v)) (@ou @v @f))
```

Implantation

Comme pour toutes les fonctions dont la donnée est un mot du langage, la forme générale de la définition de `ebs->sexpr-pref` est une conditionnelle qui passe en revue toutes les branches de la règle donnée pour l'axiome (rappelons que c'est la première règle) :

```
(define (ebs->sexpr-pref exp)
  (cond ((ebs-atomique? exp)
         ... à faire
        ((ebs-unaire? exp)
         ... à faire
         ... à faire
        ((ebs-binaire? exp)
         ... à faire
         ... à faire
         ... à faire
        (else (erreur 'ebs->sexpr-pref
                      "expression mal formée"))))
```

et il n'y a plus qu'à remplir les trous. Lorsque l'expression est réduite à une constante, la forme préfixe est l'expression elle-même :

```
(define (ebs->sexpr-pref exp)
  (cond ((ebs-atomique? exp)
         exp)
        ((ebs-unaire? exp)
         ... à faire
         ... à faire
        ((ebs-binaire? exp)
         ... à faire
         ... à faire
```

```
        ... à faire
        (else (erreur 'ebs->sexpr-pref
                      "expression mal formée")))
```

lorsque l'expression donnée est une expression unaire, sa forme préfixe est obtenue en écrivant l'opérateur unaire suivi de la forme préfixe de la sous-expression, le tout entre parenthèses (qui sont données par la liste) :

```
(define (ebs->sexpr-pref exp)
  (cond ((ebs-atomique? exp)
         exp)
        ((ebs-unaire? exp)
         (list (ebs-unaire-operateur exp)
               (ebs->sexpr-pref (ebs-unaire-operande exp))))
        ((ebs-binaire? exp)
         ... à faire
         ... à faire
         ... à faire
        (else (erreur 'ebs->sexpr-pref
                      "expression mal formée"))))
```

lorsque l'expression donnée est une expression binaire, sa forme préfixe est obtenue en écrivant l'opérateur binaire suivi de la forme préfixe de la sous-expression gauche suivi de la forme préfixe de la sous-expression droite, le tout entre parenthèses (qui sont données par la liste) :

```
(define (ebs->sexpr-pref exp)
  (cond ((ebs-atomique? exp)
         exp)
        ((ebs-unaire? exp)
         (list (ebs-unaire-operateur exp)
               (ebs->sexpr-pref (ebs-unaire-operande exp))))
        ((ebs-binaire? exp)
         (list (ebs-binaire-operateur exp)
               (ebs->sexpr-pref (ebs-binaire-operande-gauche exp))
               (ebs->sexpr-pref (ebs-binaire-operande-droit exp))))
        (else (erreur 'ebs->sexpr-pref
                      "expression mal formée"))))
```

4.1.2. Conversion en polonaise préfixé

Une expression en polonaise préfixé n'étant pas parenthésée, le résultat de la fonction permettant l'affichage n'est pas une Sexpression, mais une chaîne de caractères et nous nommerons `ebs->string-pol-pref` cette fonction.

Spécification

Sa spécification est :

```
;;; ebs->string-pol-pref : ExprBoolSimple -> Sexpression
;;; (ebs->string-pol-pref exp) rend la Sexpression représentant exp en
;;; notation préfixée, complètement parenthésée.
```

Exemple d'application

```
(let ((expression
       (ebs-binaire (ebs-unaire (ebs-operateur1-non)
                                (ebs-binaire (ebs-constante-vrai)
                                             (ebs-operateur2-et)
                                             (ebs-constante-vrai)))
                    (ebs-operateur2-et)
                    (ebs-binaire (ebs-constante-vrai)
                                 (ebs-operateur2-ou)
                                 (ebs-constante-faux)))))
  (ebs->string-pol-pref expression))
  → "@et @non @et @v @v @ou @v @f"
```

Tout d'abord, notez que, dans la notation polonaise, comme nous pouvons écrire les variables avec plusieurs caractères, les différents éléments doivent être isolés. En effet, par exemple, $\vee abc$ (ou `@ouabc`) est ambigu (est-ce $\vee a\ bc$ ou $\vee ab\ c$?).

L'implantation ne pose pas de problème :

```
;;; ebs->string-pol-pref : ExprBoolSimple -> Sexpression
;;; (ebs->string-pol-pref exp) rend la Sexpression représentant exp en
;;; notation préfixée, complètement parenthésée.
(define (ebs->string-pol-pref exp)
  (cond ((ebs-atomique? exp)
         (symbol->string exp))
        ((ebs-unaire? exp)
         (string-append (symbol->string (ebs-unaire-operateur exp))
                        " "
                        (ebs->string-pol-pref (ebs-unaire-operande exp))))
        ((ebs-binaire? exp)
         (string-append
          (symbol->string (ebs-binaire-operateur exp))
          " "
          (ebs->string-pol-pref (ebs-binaire-operande-gauche exp))
          " "
          (ebs->string-pol-pref (ebs-binaire-operande-droit exp))))
        (else (erreur 'ebs->string-pol-pref
                      "expression mal formée"))))
```

4.2. Fonctions de conversion d'entrée

Considérons, par exemple, la fonction `ebs-exp-cste?` qui prend comme donnée un élément du type `ExprBoolSimple` . Si nous voulons faire évaluer une expression d'un langage $expBoolSimple_\alpha$, nous devons avoir une fonction, α`->ebs` , qui a comme donnée un mot de $expBoolSimple_\alpha$ et qui rend une valeur du type `ExpBoolSimple` (*i.e.* un élément de la barrière syntaxique des expressions booléennes simples) :

```
;;; α->ebs: α -> ExpBoolSimple
;;; (α->ebs d) rend l'élément de ExpBoolSimple représenté,
;;; dans le langage expBoolSimple_α, par «d»
```

et, pour savoir si l'expression `exp-`α, écrite dans le langage $expBoolSimple_\alpha$, est une expression constante, il suffit d'écrire :

```
(ebs-exp-cste? (α->ebs exp-α))
```

Comme exemples, spécifions et implantons deux convertisseurs d'entrée pour les expressions booléennes simples.

4.2.1. sexpr-pref->ebs

Commençons par le langage préfixe complètement parenthésé. Un mot de ce langage étant complètement parenthésé, il peut être vu comme une Sexpression et le convertisseur d'entrée a donc comme type `S-Expression -> ExprBoolSimple` . Pour indiquer le type de la donnée et le genre (préfixe) de langage, nous nommerons `sexpr-pref->ebs` cette fonction dont la spécification est donc :

```
;;; sexpr-pref->ebs : Sexpression -> ExprBoolSimple
;;; (sexpr-pref->ebs s) rend l'arbre de syntaxe abstraite dont la représentation,
;;; dans le langage exprBoolSimple_prefixe est la Sexpression s
```

En voici une application :

```
(ebs-exp-cste? (sexpr-pref->ebs '(@ou (@et a @f) (@non @f))))
```

Pour implanter cette fonction, on analyse la Sexpression à lire :

- soit elle est réduite à un symbole (ce n'est pas une liste) et
  - soit c'est `@v` ou `@f` et il suffit de fabriquer l'expression booléenne représentée par cette constante,

– sinon, c'est une variable et il suffit de fabriquer l'expression booléenne représentée par cette variable,
  – soit c'est une Sexpression de longueur 2 et il suffit de fabriquer l'expression composée unaire dont
    – l'opérateur est représenté par le premier élément de la Sexpression donnée,
    – l'opérande est obtenue en lisant (appel récursif) le deuxième élément de la Sexpression donnée,
  – soit c'est une Sexpression de longueur 3 et il suffit de fabriquer l'expression composée binaire dont
    – l'opérande gauche est obtenue en lisant (appel récursif) le deuxième élément de la Sexpression donnée,
    – l'opérateur est représenté par le premier élément de la Sexpression donnée,
    – l'opérande droit est obtenue en lisant (appel récursif) le troisième élément de la Sexpression donnée.

D'où l'implantation :

```
(define   (sexpr-pref->ebs     s)
  (if  (pair?   s)
       (if  (pair?   (cdr   s))
            (if  (pair?   (cddr   s))
                 (if  (pair?   (cdddr   s))
                                          ; la liste a au moins 4 éléments:
                      (erreur   'sexpr-pref->ebs     s " mal  formée   (+4)")
                                          ; la liste a 3 éléments:
                      (ebs-binaire    (sexpr-pref->ebs     (cadr   s))
                                      (sexpr-pref-operateur2       (car   s))
                                      (sexpr-pref->ebs     (caddr   s))))
                                          ; la liste a 2 éléments:
                 (ebs-unaire    (sexpr-pref-operateur1       (car   s))
                                (sexpr-pref->ebs     (cadr   s))))
                                          ; la liste a 1 élément:
            (erreur   'sexpr-pref->ebs     s " mal  formée   (1)"))
       (if  (sexpr-pref-constante?       s)
            (sexpr-pref-constante       s)
            (sexpr-pref-variable       s))))

;;;  sexpr-pref-constante? : Symbole -> bool
;;;  (sexpr-pref-constante? s) rend #t ssi s représente une constante
(define   (sexpr-pref-constante?       s)
  (member   s '(@v  @f)))
;;;  sexpr-pref-constante : Symbole -> ExprBoolSimple
;;;  (sexpr-pref-constante s) rend l'expression réduite à la constante
;;;  représentée par s
(define   (sexpr-pref-constante       s)
  (cond  ((equal?   s '@v)  (ebs-constante-vrai))
         ((equal?   s '@f)  (ebs-constante-faux))
         (else  (erreur   'sexpr-pref->ebs     s "n'est   pas  une  constante"))))
;;;  sexpr-pref-variable : Symbole -> ExprBoolSimple
;;;  (sexpr-pref-variable s) rend l'expression réduite à la variable
;;;  représentée par s
(define   (sexpr-pref-variable       s)
  (ebs-variable     s))

;;;  sexpr-pref-operateur1 : Symbole -> Operateur1
;;;  (sexpr-pref-operateur1 s) rend l'opérateur unaire représenté par s
(define   (sexpr-pref-operateur1       s)
  (if  (equal?   s '@non)
       (ebs-operateur1-non)
       (erreur   'sexpr-pref->ebs     s "n'est   pas  un  opérateur    unaire")))
;;;  sexpr-pref-operateur2 : Symbole -> Operateur2
;;;  (sexpr-pref-operateur2 s) rend l'opérateur binaire représenté par s
(define   (sexpr-pref-operateur2       s)
  (cond  ((equal?   s '@et)  (ebs-operateur2-et))
         ((equal?   s '@ou)  (ebs-operateur2-ou))
         (else  (erreur   'sexpr-pref->ebs     s "n'est   pas  un  opérateur    binaire"))))
```

4.2.2. polonaise-prefixe->ebs

Donnons la grammaire :

<expBoolSimple$_{polonaise}$> → *<atomique>*
<unaire>
<binaire>

<unaire> → *<opérateur1>* *<expBoolSimple$_{polonaise}$>*

<binaire> → *<opérateur2>* *<expBoolSimple$_{polonaise}$>* *<expBoolSimple$_{polonaise}$>*

<atomique> → *<constante>*
<variable>

<constante> → `@f` FAUX
`@v` VRAI

<variable> → *Une suite de caractères autres que l'espace et les parenthèses*

<opérateur1> → `@non`

<opérateur2> → `@et` ET
`@ou` OU

Exemple : @et@non@ou@f@v@f

Premier problème :

L'écriture d'un convertisseur d'entrée est alors beaucoup moins simple que précédemment. En effet `@et @non @ou @f @v @f` n'est pas une Sexpression et nous avons vu qu'en Scheme seules les Sexpressions pouvaient être des données.

Il faut donc tricher. Deux solutions :

- entourer la donnée par des parenthèses (car on a alors une liste, cas particulier d'une Sexpression) ; l'application du convertisseur d'entrée est alors
  `(polonaise-prefixe->ebs '(@et @non @ou @f @v @f))`
- considérer la donnée comme une chaîne de caractères (rappelons que les chaînes de caractères sont des Sexpressions « atomiques ») ; l'application du convertisseur d'entrée est alors
  `(polonaise-prefixe->ebs "@et @non@ou @f @v @f")`

la seconde solution étant celle où l'on triche le moins, mais il semble que la première solution soit plus simple à traiter, aussi regardons cette solution.

Second problème :

Mais il y a un autre problème ! Pour analyser la liste (@et @non @ou @f @v @f), il faut la découper (on peut facilement voir qu'il s'agit d'un « binaire ») en

- son opérateur, `@et` : facile, c'est le `car` de la liste ;
- son premier opérande (`@non @ou @f @v`) : comment faire cela ? diffi cile...
- son second opérande (`@f`) ...

Ainsi, même en trichant au maximum, la convertion d'une expression écrite en polonaise préfi xée est très diffi cile. Aussi nous ne le ferons pas... (bien qu'étant hors programme de ce cours, un convertisseur, ayant comme donnée une chaîne de caractères, est fourni en annexe).

Noter que la conversion des expressions écrites « normalement » est encore plus diffi cile puisque, en plus du découpage, il faut gérer les priorités des opérateurs. Pour les expressions booléennes, nous vous fournirons un convertisseur des expressions écrites « normalement » (sans vous donner le source car on s'éloigne beaucoup trop du programme de ce cours).

5. Implantation de la barrière syntaxique

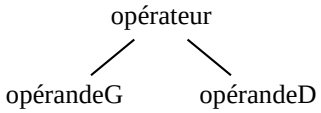
[♠♠♠ À revoir entièrement (arbres bornés 2 ? Sexpression ?) ♠♠♠]

Nous nous intéressons maintenant à l'implantation de la barrière syntaxique.

On peut implanter l'arbre de syntaxe abstraite des expressions booléennes simples à l'aide des Sexpressions :

- une constante est tout simplement implantée par le symbole Scheme correspondant,
- un opérateur est tout simplement implanté par le symbole Scheme correspondant,
- une expression unaire est implantée par une Sexpression ayant deux éléments, la Sexpression implantant l'opérateur et la Sexpression implantant l'opérande.

Une expression binaire est implantée par une Sexpression ayant trois éléments, les Sexpressions implantant, dans l'ordre, l'opérande gauche, puis l'opérateur et enfin l'opérande droit.



et nous décidons – mais nous pourrions faire un autre choix – que le premier élément de la liste correspond à l'opérande gauche (qui est une expression), que le second élément correspond à l'opérateur et que le dernier élément correspond à l'opérande droit (qui est une expression).

⇒ `(opérandeG opérateur opérandeD)`

Remarque : il faut bien comprendre que ceci n'est pas un arbre borné-deux puisque les deux sous-arbres et la racine sont mis sur le même plan : ce n'est qu'une implantation, parmi d'autres, de l'arbre de syntaxe abstraite.

Les définitions des différentes fonctions de la barrière syntaxique s'écrivent alors facilement (nous ne l'étudierons donc pas, nous vous le donnons en annexe).

5.1. sexpr-infixe->ebs

Et maintenant, transgressons nos règles ! ! !

Pour savoir si l'expression booléenne simple `exp` écrite en infixé, complètement parenthésée, est une expression constante, en utilisant la forme générale que nous avons donnée page 19, nous écririons :

```
(ebs-exp-cste?  (sexpr-infixe->ebs  exp))
```

Mais, rappelons que nous avons implanté l'arbre de syntaxe abstraite avec une Sexpression en prenant, dans le cas d'une expression binaire, comme premier élément l'opérande gauche, comme deuxième élément l'opérateur et comme troisième élément l'opérande droit. Considérons alors le langage $expBoolSimple_{infixe}$: quelle que soit sa donnée, la fonction `sexpr-infixe->ebs` rend la donnée elle-même (on dit que c'est la fonction *identité*).

Pour savoir si l'expression booléenne simple `exp` , écrite dans le langage $expression_{infixe}$, il suffit donc d'écrire, en traversant la barrière syntaxique :

```
(ebs-exp-cste?  exp)
```

Par exemple :

```
;;; exemple d'application de ebs-exp-cste?:
(ebs-exp-cste?  '((@non  (@v  @et  @v))  @ou  @f))  →#t
```

De même, pour afficher une expression en infixé, complètement parenthésée, il suffit de faire afficher, en traversant encore la barrière syntaxique, par l'interprète Scheme, la valeur de l'expression dans l'implantation infixe de la barrière syntaxique.

5.2. Retour sur les expressions en préfixé

Dans les paragraphes précédents, nous avons considéré que l'on voulait utiliser plusieurs sortes d'expressions (en infixe, suffixe...) en même temps. Dans la réalité, le plus souvent, on n'utilise qu'une notation et on s'y tient (au moins un certain temps...). Nous avons vu aussi qu'il était alors plus simple d'avoir une implantation de la barrière syntaxique en accord avec la notation retenue. Ainsi, dans l'exemple, nous avions décidé de prendre une notation infixe, aussi avons-nous implanté la barrière syntaxique en utilisant des Sexpressions où l'opérateur est en position centrale. Mais on peut changer d'avis... Si, après avoir décidé d'utiliser une notation infixe, nous décidons de n'utiliser que des expressions préfixe, au lieu d'écrire des convertisseurs d'entrée et de sortie, il y a plus simple, plus sûr et plus efficace : changer l'implantation de la barrière d'abstraction en utilisant une implantation préfixe (et, ainsi, les convertisseurs deviennent inutiles). Voici le source des seules définitions à modifier (la définiton de `ebs-binaire-operande-droite` restant la même par hazard) :

```
;;; ebs-binaire-operande-gauche : Binaire -> ExprBoolSimple
;;; (ebs-binaire-operande-gauche exp) rend l'expression, opérande gauche de
;;; l'expression donnée
(define  (ebs-binaire-operande-gauche  exp)
```

```
  (cadr exp))

;;; ebs-binaire-operateur : Binaire -> Operateur2
;;; (ebs-binaire-operateur exp) rend l'opérateur (principal) de l'expression donnée
(define (ebs-binaire-operateur exp)
  (car exp))

;;; ebs-binaire : ExprBoolSimple * Operateur2 * ExprBoolSimple -> ExprBoolSimple
;;; (ebs-binaire exp1 op exp2) rend l'expression composée ayant comme
;;; opérande gauche exp1, comme opérateur op et comme opérande droit exp2
(define (ebs-binaire exp1 op exp2)
  (list op exp1 exp2))
```

5.3. Conclusion

Aussi, dans la plupart des cas, nous tricherons :
- en prenant un langage complètement parenthésée (*i.e.* une Sexpression),
- en implantant l'arbre de syntaxe abstraite avec la même forme que le langage considéré,
- et si nous voulons changer de forme de représentation (infixe, préfixe, postfixe...), nous modifierons carrément l'implantation de la barrière d'abstraction.

6. Barrière d'interprétation

6.1. Barrières d'interprétation

Si l'on veut évaluer les expressions, encore faut-il savoir à quoi correspondent les terminaux. Ainsi, dans notre exemple, il faut savoir que les symboles `@v` et `@f` correspondent aux valeurs de vérité – mais quelles valeurs de vérité ? – et que les symboles `@ou` , `@et` , et `@non` correspondent respectivement à la disjonction, la conjonction et la négation.

Ainsi doit-on utiliser cinq fonctions, qui constituent la barrière d'interprétation du langage :

```
;;; faux : -> Booleen
;;; (faux) rend la valeur de vérité « faux »

;;; vrai : -> Booleen
;;; (vrai) rend la valeur de vérité « vrai »

;;; non : Booleen -> Booleen
;;; (non b) rend la négation de b dans le type Booleen

;;; et : Booleen * Booleen -> Booleen
;;; (et b1 b2) rend la conjonction de b1 et de b2 dans le type Booleen

;;; ou : Booleen * Booleen -> Booleen
;;; (ou b1 b2) rend la disjonction de b1 et de b2 dans le type Booleen
```

Implantations de la barrière d'interprétation

Donnons une première implantation, dans les booléens Scheme :

```
;;;; Interprétation avec Booleen = #f, #t
(define (faux)
  #f)
(define (vrai)
```

```
#f)
(define (faux? b)
  (not b))
(define (vrai? b)
  b)
(define (non b)
  (not b))
(define (et b1 b2)
  (and b1 b2))
(define (ou b1 b2)
  (or b1 b2))
```

Noter l'implantation des opérations qui sont des fonctions qui retournent une fonction.

Une autre implantation, en prenant comme valeurs booléennes les symboles `@v` et `@f` :

```
;;;; Interprétation avec Booleen = @f, @v
(define (faux) '@f)

(define (vrai) '@v)

(define (faux? b)
  (equal? b '@f))

(define (vrai? b)
  (equal? b '@v))

(define (non b)
  (if (equal? b '@f) '@v '@f))

(define (et b1 b2)
  (if (and (equal? b1 '@v) (equal? b2 '@v)) '@v '@f))

(define (ou b1 b2)
  (if (or (equal? b1 '@v) (equal? b2 '@v)) '@v '@f))
```

6.2. Évaluation des expressions constantes

Nous avons spécifié toutes les fonctions nécessaires à l'écriture de la fonction `ebs-eval-expr-cste` qui a comme spécification :

```
;;; ebs-eval-expr-cste : ExprBoolSimple -> Booleen
;;; (ebs-eval-expr-cste exp) rend la valeur de vérité de l'expression exp
```

En effet, voici une définition de cette fonction :

```
(define (ebs-eval-expr-cste exp)
  (cond ((ebs-constante? exp)
         (ebs-eval-constante exp))
        ((ebs-unaire? exp)
         (ebs-eval-unaire exp))
        ((ebs-binaire? exp)
         (ebs-eval-binaire exp))
        (else (erreur 'ebs-eval-cste "expression mal formée"))))

;;; ebs-eval-constante : Constante -> Booleen
;;; (ebs-eval-constante cste) rend la valeur de vérité de la constante cste
(define (ebs-eval-constante cste)
  (if (ebs-constante-vrai? cste)
```

```
      (vrai)
      (faux)))
;;; ebs-eval-unaire : Unaire -> Booleen
;;; (ebs-eval-unaire exp) rend la valeur de vérité de l'expression unaire exp
(define (ebs-eval-unaire exp)
  (let ((operateur (ebs-unaire-operateur exp))
        (operande (ebs-unaire-operande exp)))
    (if (ebs-operateur1-non? operateur)
        (non (ebs-eval-expr-cste operande))
        (erreur 'ebs-eval-unaire operateur "n'est pas un opérateur unaire"))))

;;; ebs-eval-binaire : Binaire -> Booleen
;;; (ebs-eval-binaire exp) rend la valeur de vérité de l'expression binaire exp
(define (ebs-eval-binaire exp)
  (let ((operateur (ebs-binaire-operateur exp))
        (operandeG (ebs-binaire-operande-gauche exp))
        (operandeD (ebs-binaire-operande-droit exp)))
    (cond ((ebs-operateur2-et? operateur)
           (et
            (ebs-eval-expr-cste operandeG)
            (ebs-eval-expr-cste operandeD)))
          ((ebs-operateur2-ou? operateur)
           (ou
            (ebs-eval-expr-cste operandeG)
            (ebs-eval-expr-cste operandeD)))
          (else (erreur 'ebs-eval-binaire operateur
                        "n'est pas un opérateur binaire")))))
```

Remarque très importante : noter que cette définition est valable quel que soit le langage utilisé (préfixe, infixe...).

7. Transformations d'expressions booléennes simples

Nous voudrions écrire la fonction `ebs-simplifiee` qui évalue partiellement une expression booléenne simple en éliminant, au maximum, les constantes booléennes.

7.1. Spécification

Ainsi cette fonction a comme donnée une expression booléenne simple (avec variables) et comme résultat une autre expression, simplifiée de l'expression donnée :

```
;;; ebs-simplifiee : ExprBoolSimple -> ExprBoolSimple
;;;  ERREUR lorsque exp n'est pas une expression bien formée
;;; (ebs-simplifiee expr) rend une expression booléenne simple simplifiée
;;; équivalente à l'expression booléenne simple donnée.
```

Exemple :

```
(ebs-simplifiee '((@non a) @et (b @ou (@v @et a))))
→ ((@non a) @et (b @ou a))
(ebs-simplifiee '((@non @f) @et (b @ou (@v @et a))))
→ (b @ou a)
(ebs-simplifiee '((@non @f) @et (b @ou (@v @ou a))))
→ @v
```

Règles de simplification :

Encore faut-il préciser ce que nous entendons par « simplifiée ». Nous utiliserons les simplifications suivantes :

(@v @ou a) ⇒ @v	(a @et @v) ⇒ a	(non @v) ⇒ @f
(a @ou @f) ⇒ a	(a @et @f) ⇒ @f	(non @f) ⇒ @v
(@v @ou a) ⇒ @v	(@v @et a) ⇒ a	
(@f @ou a) ⇒ a	(@f @et a) ⇒ @f	

Remarques :

1. il s'agit bien d'une évaluation partielle, d'autant plus que les sous-expressions qui n'ont pas d'occurrences de variables sont évaluées ;
2. nous pourrions donner d'autres règles de simplification (par exemple (@non (@non a)) ⇒a). Vous pouvez essayer d'implanter cette dernière règle de simplification – et, éventuellement, d'autres. Nous nous en tiendrons là.

7.2. Implantation

Encore le même schéma :

```
(define (ebs-simplifiee exp)
  (cond ((ebs-atomique? exp)

         ((ebs-unaire? exp)


         ((ebs-binaire? exp)



         (else (erreur 'ebs-simplifiee "expression mal formée"))))
```

une formule atomique est simplifiée ; on la retourne donc telle quelle :

```
(define (ebs-simplifiee exp)
  (cond ((ebs-atomique? exp)
         exp)
         ((ebs-unaire? exp)


         ((ebs-binaire? exp)



         (else (erreur 'ebs-simplifiee "expression mal formée"))))
```

Pour les expressions unaires et binaires, cela semble plus compliqué : nous utilisons des fonctions `ebs-unaire-simplifiee` et `ebs-binaire-simplifiee` qui simplifient des expressions unaires et binaires, mais données par leurs décompositions :

```
(define (ebs-simplifiee exp)
  (cond ((ebs-atomique? exp)
         exp)
         ((ebs-unaire? exp)
          (ebs-unaire-simplifiee (ebs-unaire-operateur exp)
                                 (ebs-unaire-operande exp)))
         ((ebs-binaire? exp)
          (ebs-binaire-simplifiee (ebs-binaire-operateur exp)
                                  (ebs-binaire-operande-gauche exp)
                                  (ebs-binaire-operande-droit exp)))
         (else (erreur 'ebs-simplifiee "expression mal formée"))))
```

avec ces fonctions spécifiées par :

```
;;; ebs-unaire-simplifiee : Operateur1 * ExprBoolSimple -> ExprBoolSimple
;;; (ebs-unaire-simplifiee op exp) rend une expression booléenne simple
```

```
;;; simplifiée équivalente à l'expression booléenne (ebs-unaire op exp).

;;; ebs-binaire-simplifiee : Operateur2 * ExprBoolSimple * ExprBoolSimple -> ExprBoolSimple
;;; (ebs-binaire-simplifiee op expG expD) rend une expression booléenne
;;; simple simplifiée équivalente à l'expression booléenne
;;; (ebs-binaire expG op expD).
```

7.2.1. Implantation de `ebs-unaire-simplifiee`

Pour la simplification d'une expression unaire, on vérifie que l'opérateur est bien `@non` et, si c'est bien le cas, on fait appel à la fonction de simplification dédiée à la négation. En remarquant que la simplification d'une négation doit être calculée à partir de la simplification de sa sous-expression :

```
;;; ebs-unaire-simplifiee : Operateur1 * ExprBoolSimple -> ExprBoolSimple
;;; (ebs-unaire-simplifiee op exp) rend une expression booléenne simple
;;; simplifiée équivalente à l'expression booléenne (ebs-unaire op exp).
(define (ebs-unaire-simplifiee op exp)
  (if (ebs-operateur1-non? op)
      (ebs-neg-simplifiee (ebs-simplifiee exp))
      (erreur 'ebs-simplifiee op "n'est pas un opérateur unaire")))
```

en utilisant une fonction, `ebs-neg-simplifiee` qui doit avoir comme spécification :

```
;;; ebs-neg-simplifiee : ExprBoolSimple/simplifiee/ -> ExprBoolSimple/simplifiee/
;;; (ebs-neg-simplifiee exp) rend une expession booléenne simple simplifiée
;;; équivalente à (non exp)
```

7.2.2. Implantation de `ebs-binaire-simplifiee`

De même, pour simplifier une expression binaire, on fait appel à une fonction de simplification dédiée à l'opérateur de l'expression :

```
(define (ebs-binaire-simplifiee op expG expD)
  (cond ((ebs-operateur2-et? op)
         (ebs-conj-simplifiee (ebs-simplifiee expG) (ebs-simplifiee expD)))
        ((ebs-operateur2-ou? op)
         (ebs-disj-simplifiee (ebs-simplifiee expG) (ebs-simplifiee expD)))
        (else (erreur 'ebs-simplifiee op "n'est pas un opérateur binaire"))))
```

ces fonctions ayant comme spécification :

```
;;; ebs-conj-simplifiee : ExprBoolSimple/simplifiee/ * ExprBoolSimple/simplifiee/
;;;   -> ExprBoolSimple/simplifiee/
;;; (ebs-conj-simplifiee e1 e2) rend une expession booléenne simple simplifiée
;;; équivalente à (e1 et e2)
```

```
;;; ebs-disj-simplifiee : ExprBoolSimple/simplifiee/ * ExprBoolSimple/simplifiee/
;;;   -> ExprBoolSimple/simplifiee/
;;; (ebs-disj-simplifiee e1 e2) rend une expession booléenne simple simplifiée
;;; équivalente à (e1 ou e2)
```

7.2.3. Implantation de `ebs-neg-simplifiee`

Règles de simplification (rappel) :

Rappelons les règles de simplification pour la négation :

(@non @v) ⇒ @f
(@non @f) ⇒ @v

ce à quoi on pourrait ajouter, pour tenir compte des cas où il n'y a pas de simplification :

(@non a) ⇒ (@non a)

Rappelons la spécification de la fonction `ebs-neg-simplifiee` :

```
;;; ebs-neg-simplifiee : ExprBoolSimple/simplifiee/ -> ExprBoolSimple/simplifiee/
;;; (ebs-neg-simplifiee exp) rend une expession booléenne simple simplifiée
;;; équivalente à (non exp)
```

et notons que la donnée de cette fonction est une expression simplifiée et que c'est la sous-expression de la négation (c'est le `a` de la règle ci-dessus). L'implantation est alors facile (noter tout de même l'utilisation de la barrière syntaxique) :

```
(define (ebs-neg-simplifiee exp)
  (cond ((ebs-constante-vrai? exp) (ebs-constante-faux))
        ((ebs-constante-faux? exp) (ebs-constante-vrai))
        (else (ebs-unaire (ebs-operateur1-non) exp))))
```

7.2.4. Implantation de `ebs-conj-simplifiee`

Règles de simplification (rappel) :

Rappelons les règles de simplification pour la conjonction :

(@f @et a) ⇒ @f	(a @et @f) ⇒ @f
(@v @et a) ⇒ a	(a @et @v) ⇒ a

en se rappelant que la donnée de la fonction `ebs-conj-simplifiee` est constituée par les deux sous-expressions simplifiée d'une conjonction, l'implantation de cette fonction est :

```
;;; ebs-conj-simplifiee : ExprBoolSimple/simplifiee/ * ExprBoolSimple/simplifiee/
;;;   -> ExprBoolSimple/simplifiee/
;;; (ebs-conj-simplifiee e1 e2) rend une expession booléenne simple simplifiée
;;; équivalente à (e1 et e2)
(define (ebs-conj-simplifiee e1 e2)
  (cond ((ebs-constante-faux? e1) (ebs-constante-faux))
        ((ebs-constante-faux? e2) (ebs-constante-faux))
        ((ebs-constante-vrai? e1) e2)
        ((ebs-constante-vrai? e2) e1)
        (else (ebs-binaire e1 (ebs-operateur2-et) e2))))
```

Idem pour `ebs-disj-simplifiee` qui est laissée en exercice.

8. Notion d'environnement

Nous voudrions évaluer les expressions booléennes simples ayant des variables. Mais pour ce faire, il faut associer une valeur à chaque variable. Autrement dit, le problème est d'évaluer une expression booléenne bien formée dans un **environnement** qui est représenté par une liste d'associations `(clef valeur)` . Par exemple,

env : a ⇝ @v
b ⇝ @f

si, dans `env` , la valeur de `a` est `@v` et la valeur de `b` est `@f` :

`(ebs-eval '((@non a) @et (b @ou (@v @et a))) env)` → `@f`

8.0.5. Spécification de `ebs-eval`

```
;;; ebs-eval : ExprBoolSimple * Environnement -> Booleen
;;; (ebs-eval exp env) rend la valeur de vérité de l'expression exp dans
```

;;; *l'environnement env*

8.0.6. Création d'un environnement

Il faut que l'on puisse créer un environnement qui associe à chaque variable une valeur. Pour ce faire, nous pourrions définir une fonction qui aurait comme donnée une liste d'associations variable — valeur et qui créerait l'environnement voulu. Mais, la plupart du temps, lorsqu'on travaille sous un environnement, au départ, on ne part pas de zéro, mais, au contraire, dans un environnement initial. Aussi, pour créer des environnements, préfère-t-on avoir deux fonctions : la première – `env-initial` – rend l'environnement initial et la seconde – `env-ajouts` – permet de rajouter des associations variable — valeur. Ainsi, l'exemple précédent s'écrira :

```
(let((env   (env-ajouts   (list  (list  'a (vrai))
                                 (list  'b (faux)))
                          (env-initial))))
  (ebs-eval   '((@non   a) @et  (b @ou  (@v  @et  a)))  env))
```

8.0.7. Implantation de `ebs-eval`

La définition de la fonction `ebs-eval` suit toujours le même schéma. Nous ne le détaillons pas :

```
(define   (ebs-eval   exp  env)
  (cond  ((ebs-atomique?     exp)
          (ebs-eval-atomique      exp  env))
         ((ebs-unaire?     exp)
          (ebs-eval-unaire      exp  env))
         ((ebs-binaire?     exp)
          (ebs-eval-binaire      exp  env))
         (else  (erreur   'ebs-eval   "expression    mal  formée"))))
```

Les fonctions `non` , `et` et `ou` sont des fonctions de la barrière d'interprétation que nous avons déjà utilisées pour évaluer une expression constante. Reste la fonction `atomique-val` qui doit avoir comme spécification :

```
;;;  ebs-eval-atomique : Atomique * Environnement -> Booleen
;;;  (ebs-eval-atomique exp env) rend la valeur de vérité de l'expression
;;;  atomique exp dans l'environnement env
```

8.0.8. Première implantation de `ebs-eval-atomique`

Il suffit, une fois de plus, de suivre la grammaire : une expression atomique est une constante ou une variable :

```
;;;  ebs-eval-atomique : Atomique * Environnement -> Booleen
;;;  (ebs-eval-atomique exp env) rend la valeur de vérité de l'expression
;;;  atomique exp dans l'environnement env
(define   (ebs-eval-atomique      exp  env)
  (if  (ebs-constante?     exp)
       (if  (ebs-constante-vrai?      exp)
            (vrai)
            (faux))
       (env-variable-val      exp  env)))
```

et

- pour une constante, c'est soit la constante vraie, soit la constante faux et il suffit d'utiliser les fonctions (de la barrière d'interprétation) `vrai` et `faux` ,
- pour une variable, nous rajoutons, dans la barrière d'abstraction des environnements, la fonction `env-variable-val` de spécification :

```
;;;  env-variable-val : Variable * Environnement[Variable Domaine] -> Domaine
;;;  ERREUR lorsque exp n'est pas définie dans l'environnement
;;;  (env-variable-val exp env) rend la valeur de exp dans l'environnement env
```

Barrière d'abstraction des environnements

Rappelons les différentes fonctions que nous avons mises sur les environnements en remarquant que l'environnement initial est vide :

```
;;; env-initial : -> Environnement[Variable Domaine]
;;; (env-initial) rend l'environnement vide.
```

```
;;; env-ajouts : LISTE[N-UPLET[Variable Domaine]] * Environnement[Variable Domaine]
;;;              -> Environnement[Variable Domaine]
;;; (env-ajouts L env) rend l'environnement obtenu en étendant
;;; l'environnement env en associant à chaque variable de la liste
;;; d'associations L, la valeur correspondante.
```

```
;;; env-variable-val : Variable * Environnement[Variable Domaine] -> Domaine
;;; ERREUR lorsque exp n'est pas défi nie dans l'environnement
;;; (env-variable-val exp env) rend la valeur de exp dans l'environnement env
```

Implantation de la barrière d'abstraction des environnements

Nous n'étudierons pas l'implantation qui est facile (en fait, nous avons déjà vu de telles implantations lorsque nous avons étudié les listes d'associations) ; elle vous est donnée en annexe.

8.0.9. Seconde implantation de `ebs-eval-atomique`

En fait, il s'agit plutôt d'une seconde vision de l'environnement : au lieu de tester si l'expression atomique est une constante ou une variable, on peut mettre la valeur des constantes dans l'environnement initial et le calcul de `ebs-eval-atomique` est alors tout simplement le calcul de `env-variable-val` de la solution précédente. Ainsi, la fonction `ebs-eval-atomique` n'existe plus et la fonction `ebs-eval-atomique` , renommée `env-atomique-val` , est une fonction de la barrière d'abstraction des environnements.

Barrière d'abstraction des environnements

La barrière d'abstraction des environnements est donc :

```
;;; env-initial : -> Environnement[Symbole Booleen]
;;; (env-initial) rend l'environnement associant aux deux constantes
;;; booléennes leurs valeurs dans Booleen
```

```
;;; env-ajouts : LISTE[N-UPLET[Variable Domaine]] * Environnement[Symbole Domaine]
;;;              -> Environnement[Symbole Domaine]
;;; (env-ajouts L env) rend l'environnement obtenu en étendant
;;; l'environnement env en associant à chaque variable de la liste
;;; d'associations L, la valeur correspondante.
```

```
;;; env-atomique-val : Symbole * Environnement[Symbole Domaine] -> Domaine
;;; ERREUR lorsque exp n'est pas défi nie dans l'environnement
;;; (env-atomique-val exp env) rend la valeur de exp dans l'environnement env
```

Implantation de la barrière d'abstraction des environnements

Là encore, l'implantation est facile et elle vous est donnée dans l'annexe.

8.0.10. Différence sémantique entre ces deux visions

Considérons l'environnement `env` défi ni comme suit :

```
(env-ajouts (list (list '@v (faux)))
            (env-initial))
```

— Quelle est la valeur retournée, dans les deux implantations précédentes, lorsque l'on évalue l'expression définie à la constante `@v` dans cet environnement ?

Première vision (environnement initial vide)

```
(let ((env (env-ajouts (list (list '@v (faux)))
                       (env-initial))))
  (ebs-eval '@v env)) → @v
```

Seconde vision (valeurs constantes dans environnement initial)

```
(let ((env (env-ajouts (list (list '@v (faux)))
                       (env-initial))))
  (ebs-eval '@v env)) → @f
```

Constante — variable prédéfinie

En fait, dans la première vision, `@v` et `@f` sont des constantes (leur valeur ne change jamais) alors que dans la seconde vision, `@v` et `@f` sont traités comme des variables sauf que, dès le départ, elles ont une valeur : on dit que ce sont des variables prédéfinies.

Ces notions de constantes et de variables prédéfinies se retrouvent dans tous les langages (sous des noms différents, en particulier avec la notion de mot-clef qui correspond à nos constantes. Par exemple, en Scheme, on peut utiliser n'importe quel symbole comme nom de fonction, sauf `and`, `or`, `define` ... Ainsi, on peut redéfinir `+` (qui est bien sûr prédéfini), mais on ne peut pas redéfinir `and`.

9. Expressions booléennes généralisées

9.1. Grammaire

On définit les expressions booléennes généralisées par la grammaire :

<expBoolGen> → *<atomique>*
<négation>
<n-aire>

<atomique> → *<constante>*
<variable>

<constante> → `@v` VRAI
`@f` FAUX

<variable> → *Une suite de caractères autres que l'espace, les parenthèses et @*

<négation> → `(@non` *<expBoolGen>* `)`

<n-aire> → `(@et` *<expBoolGen>** `)` CONJONCTION
`(@ou` *<expBoolGen>** `)` DISJONCTION

Remarque : comme en Scheme, les disjonctions et les conjonctions opèrent sur un nombre quelconque d'arguments, y compris 1 argument ou même 0 argument.

9.1.1. Exemples

```
(@et (@non (@et @v a)) (@ou b (@et @v a)))
(@ou a)
(@ou)
```

Lorsqu'il y a un argument, la sémantique va de soi : la valeur de vérité d'une conjonction ou d'une disjonction n'ayant qu'une sous-expression est égale à la valeur de vérité de la sous-expression. La valeur de vérité d'une disjonction sans argument est « faux » et la valeur de vérité d'une conjonction sans argument est « vrai ».

9.2. Barrières d'abstraction

9.2.1. Barrière syntaxique

```
;;;;  Reconnaisseurs :
;;;  ebg-atomique? : ExprBoolGen -> bool
;;;  ebg-constante? : ExprBoolGen -> bool
;;;  ebg-variable? : Atomique -> bool
;;;  ebg-constante-vrai? : Constante -> bool
;;;  ebg-constante-faux? : Constante -> bool
;;;  ebg-negation? : ExprBoolGen -> bool
;;;  ebg-conjonction? : ExprBoolGen -> bool
;;;  ebg-disjonction? : ExprBoolGen -> bool

;;;;  Accesseurs :
;;;  ebg-neg-sous-exp : Negation -> ExprBoolGen
;;;  ebg-n-aire-sous-exps : N-aire -> LISTE[ExprBoolGen]

;;;;  Constructeurs :
;;;  ebg-constante-vrai : -> Constante
;;;  ebg-constante-faux : -> Constante
;;;  ebg-variable : Symbole -> ExprBoolGen
;;;  ebg-negation : ExprBoolGen -> ExprBoolGen
;;;  ebg-conjonction : LISTE[ExprBoolGen] -> ExprBoolGen
;;;  ebg-disjonction : LISTE[ExprBoolGen] -> ExprBoolGen
```

9.2.2. Barrière d'interprétation

```
;;;  vrai : -> Booleen
;;;  faux : -> Booleen
;;;  vrai? : Booleen -> bool
;;;  faux? : Booleen -> bool
;;;  non : Booleen -> Booleen
;;;  et : Booleen * Booleen -> Booleen
;;;  ou : Booleen * Booleen -> Booleen
```

9.3. Évaluation d'une expression booléenne constante

9.3.1. Spécification

Exactement comme pour les expressions booléennes simples :

```
;;;  ebg-exp-const-val : ExprBoolGen/constante/ -> Booleen
;;;  ERREUR lorsque «exp» n'est pas une expression constante bien formée
;;;  (ebg-exp-const-val exp) rend la valeur de «exp»
```

Exemples :

```
(ebg-exp-const-val     '(@et  (@ou  @v @f @f)  (@non  @f)  @v))  → VRAI
(ebg-exp-const-val     '(@et  (@ou  @v @f @f)  (@non  @f)  @v  (@et  @v @f)))  → FAUX
```

9.3.2. Implantation

Comme d'habitude, le schéma de la défi nition suit la grammaire :

```
(define   (ebg-exp-const-val      exp)
  (cond   ((ebg-atomique?     exp)
           ... à faire
           ... à faire
```

```
          ... à faire
          ... à faire
          ... à faire
         ((ebg-negation?     exp)
          ... à faire
         ((ebg-conjonction?      exp)
          ... à faire
         ((ebg-disjonction?      exp)
          ... à faire
         (else  (erreur   'ebg-exp-const-val
                          "mal  formée  ou  non  constante"))))
```

pour les constantes et les négations, on fait comme pour les expressions simples :

```
(define   (ebg-exp-const-val      exp)
  (cond   ((ebg-atomique?     exp)
           (if  (ebg-constante?     exp)
                (if  (ebg-constante-vrai?      exp)
                     (vrai)
                     (faux))
                (erreur   'ebg-exp-const-val      exp  " est  une  variable")))
          ((ebg-negation?     exp)
           (non  (ebg-exp-const-val      (ebg-neg-sous-exp      exp))))
          ((ebg-conjonction?      exp)
           ... à faire
          ((ebg-disjonction?      exp)
           ... à faire
          (else  (erreur   'ebg-exp-const-val
                           "mal  formée  ou  non  constante"))))
```

En revanche, les conjonctions et les disjonctions ne peuvent pas être traitées comme dans le cas des expressions simples puisque l'on peut avoir un nombre quelconque de sous-expressions.

Calculer la valeur d'une conjonction paraît compliqué : nous spécifi ons (et nous implanterons) une fonction pour le faire :

```
(define   (ebg-exp-const-val      exp)
  (cond   ((ebg-atomique?     exp)
           (if  (ebg-constante?     exp)
                (if  (ebg-constante-vrai?      exp)
                     (vrai)
                     (faux))
                (erreur   'ebg-exp-const-val      exp  " est  une  variable")))
         ((ebg-negation?     exp)
           (non  (ebg-exp-const-val      (ebg-neg-sous-exp      exp))))
          ((ebg-conjonction?      exp)
           (ebg-conjonction-const-val        (ebg-n-aire-sous-exps       exp)))
          ((ebg-disjonction?      exp)
           ... à faire
          (else  (erreur   'ebg-exp-const-val
                           "mal  formée  ou  non  constante"))))
```

```
;;;  ebg-conjonction-const-val : LISTE[ExprBoolGen] -> Booleen
;;;  ERREUR lorsqu'un des éléments de «exps» n'est pas une expression constante bien formée
;;;  (ebg-conjonction-const-val exps) rend la valeur de la conjonction de tous les
;;;  éléments de «exps»
```

Noter que la donnée de la fonction `ebg-conjonction-const-val` est une liste d'expressions et, lors de son

application dans ebg-exp-const-val, elle est appelée avec la liste des sous-expressions de l'expression donnée.

On agit de même pour les disjonctions :

```
(define   (ebg-exp-const-val      exp)
  (cond   ((ebg-atomique?     exp)
           (if  (ebg-constante?     exp)
                (if  (ebg-constante-vrai?       exp)
                     (vrai)
                     (faux))
                (erreur   'ebg-exp-const-val      exp  " est  une  variable")))
          ((ebg-negation?     exp)
           (non  (ebg-exp-const-val      (ebg-neg-sous-exp      exp))))
           ((ebg-conjonction?      exp)
            (ebg-conjonction-const-val        (ebg-n-aire-sous-exps       exp)))
           ((ebg-disjonction?      exp)
            (ebg-disjonction-const-val        (ebg-n-aire-sous-exps       exp)))
           (else   (erreur   'ebg-exp-const-val
                             "mal  formée   ou  non  constante"))))
```

```
;;;  ebg-disjonction-const-val : LISTE[ExprBoolGen] -> Booleen
;;;  ERREUR lorsqu'un des éléments de «exps» n'est pas une expression constante bien formée
;;;  (ebg-disjonction-const-val exps) rend la valeur de la disjonction de tous les
;;;  éléments de «exps»
```

9.3.3. Implantation de `ebg-conjonction-const-val`

Rappelons la spécification :

```
;;;  ebg-conjonction-const-val : LISTE[ExprBoolGen] -> Booleen
;;;  ERREUR lorsqu'un des éléments de «exps» n'est pas une expression constante bien formée
;;;  (ebg-conjonction-const-val exps) rend la valeur de la conjonction de tous les
;;;  éléments de «exps»
```

La donnée de cette fonction étant une liste, on suit le schéma sur les listes :

```
(define   (ebg-conjonction-const-val        exps)
  (if  (pair?   exps)
       ; valeur lorsque la liste n'est pas vide
       ... à faire
       ... à faire
       ; valeur lorsque la liste est vide
)
```

Comme nous l'avons dit plus haut, la valeur de la conjonction d'une liste d'expressions vide est `VRAI` :

```
(define   (ebg-conjonction-const-val        exps)
  (if  (pair?   exps)
       ; valeur lorsque la liste n'est pas vide
       ... à faire
       ... à faire
       (vrai))
)
```

Lorsque la liste n'est pas vide, la valeur de vérité de la conjonction de ses éléments est égale à la conjonction (binaire) de la valeur de vérité de son premier élément et de la valeur de vérité de la conjonction (généralisée) de ses autres éléments. On pourait donc utiliser la fonction `et`, mais, pour des raisons d'efficacité, on préfère calculer la conjonction binaire à l'aide d'une alternative :

```
(define   (ebg-conjonction-const-val        exps)
```

```
    (if (pair? exps)
        (if (vrai? (ebg-exp-const-val (car exps)))
            (ebg-conjonction-const-val (cdr exps))
            (faux))
        (vrai)))
```

Remarque : nous pourrions aussi écrire la définition suivante :

```
(define (ebg-conjonction-const-val exps)
  (reduce et (vrai) (map ebg-exp-const-val exps)))
```

Nous avons préféré la version donnée ci-dessus car elle est plus efficace. En effet, dans cette version, dès qu'une sous-expression est fausse, on ne calcule pas la valeur de vérité des autres sous-expressions alors que dans la version avec `map` et `reduce`, on calcule toujours la valeur de vérité de toutes les sous-expressions.

9.3.4. Implantation de `ebg-disjonction-const-val`

Laissée en exercice.

9.4. Simplification d'une expression booléenne avec inconnues

9.4.1. Spécification

Exactement comme pour les expressions simples :

```
;;; ebg-exp-simplifiee : ExprBoolGen -> ExprBoolGen
;;;  ERREUR lorsque «exp» n'est pas une expression bien formée
;;; (ebg-exp-simplifiee expr) rend une expression booléenne simplifiée équivalente à
;;; l'expression booléenne donnée.
```

Exemples :

```
(ebg-exp-simplifiee '(@et @v a b))
→ (@et a b)
(ebg-exp-simplifiee '(@et (@non (@et @v a)) (@ou b (@et @v a))))
→ (@et (@non a) (@ou b a))
```

9.4.2. Implantation

Encore une fois, nous suivons le même schéma :

```
(define (ebg-exp-simplifiee exp)
  (cond
    ((ebg-atomique? exp)
     ... à faire
    ((ebg-negation? exp)
     ... à faire
    ((ebg-conjonction? exp)
     ... à faire
     ... à faire
    ((ebg-disjonction? exp)
     ... à faire
     ... à faire
    (else (erreur 'ebg-exp-simplifiee "expression mal formée"))))
```

Comme pour les expressions simples, la simplifiée d'une expression atomique est égale à elle-même :

```
(define (ebg-exp-simplifiee exp)
  (cond
    ((ebg-atomique? exp)
     exp)
    ((ebg-negation? exp)
```

```
    ... à faire
    ((ebg-conjonction?      exp)
     ... à faire
     ... à faire
    ((ebg-disjonction?      exp)
     ... à faire
     ... à faire
    (else  (erreur  'ebg-exp-simplifiee      "expression    mal  formée"))))
```

les négations se traitent exactement comme dans le cas des expressions simples, aussi nous ne les revoyons pas :

```
(define  (ebg-exp-simplifiee      exp)
  (cond
    ((ebg-atomique?     exp)
     exp)
    ((ebg-negation?     exp)
     (ebg-neg-simpl     (ebg-exp-simplifiee      (ebg-neg-sous-exp      exp))))
    ((ebg-conjonction?      exp)
     ... à faire
     ... à faire
    ((ebg-disjonction?      exp)
     ... à faire
     ... à faire
    (else  (erreur  'ebg-exp-simplifiee      "expression    mal  formée"))))
```

Pour les conjonctions (et il en va de même pour les disjonctions) on doit effectuer une opération de simplification sur les simplifiées de toutes les sous-expressions. Pour avoir la liste des simplifiées de toutes les sous-expressions, il suffit de « maper » la fonction `ebg-exp-simplifiee` sur la liste des sous-expressions :

```
(define  (ebg-exp-simplifiee      exp)
  (cond
    ((ebg-atomique?     exp)
     exp)
    ((ebg-negation?     exp)
     (ebg-neg-simpl     (ebg-exp-simplifiee      (ebg-neg-sous-exp      exp))))
    ((ebg-conjonction?      exp)
     (ebg-conj-simpl     (map  ebg-exp-simplifiee
                               (ebg-n-aire-sous-exps      exp))))
    ((ebg-disjonction?      exp)
     (ebg-disj-simpl     (map  ebg-exp-simplifiee
                               (ebg-n-aire-sous-exps      exp))))
    (else  (erreur  'ebg-exp-simplifiee      "expression    mal  formée"))))
```

la fonction `ebg-conj-simpl` ayant comme spécification :

```
;;;  ebg-conj-simpl : LISTE[ExprBoolGen/simplifiee/] -> ExprBoolGen/simplifiee/
;;;  (ebg-conj-simpl exps) rend une expression booléenne simplifiée équivalente à
;;;  la conjonction des éléments de la liste «exps»
```

(Le cas de la disjonction est similaire, nous ne le traiterons pas)

9.4.3. Implantation de `ebg-conj-simpl`

Comment peut-on simplifier une conjonction ?

- si une des sous-expressions est fausse, l'expression est fausse,
- on peut supprimer toutes les sous-expressions vraies,
- une conjonction sans sous-expressions est vraie,
- une conjonction ayant une seule sous-expression est équivalente à cette sous-expression.

D'où l'implantation de `ebg-conj-simpl` :

```
(define (ebg-conj-simpl exps)
  (if (member (ebg-constante-faux) exps)
      (ebg-constante-faux)
      (let ((exps-simp (ebg-liste-sans (ebg-constante-vrai) exps)))
        (if (pair? exps-simp)
            (if (pair? (cdr exps-simp))
                (ebg-conjonction exps-simp)
                (car exps-simp))
            (ebg-constante-vrai)))))
```

la fonction `ebg-liste-sans` ayant comme spécification :

```
;;; ebg-liste-sans : alpha * LISTE[alpha] -> LISTE[alpha]
;;; (ebg-liste-sans el ll) rend la liste des éléments de «ll» qui ne sont pas égaux à «el»
```

L'implantation de cette fonction, simple exercice sur les listes, est laissée en exercice (on pourra utiliser la fonction `filtre`).

Quatrième saison

Version 1.2

Sommaire

1. Spécification de deug-eval

1.1. Introduction

L'évaluation convertit un texte en une valeur. Ainsi, le cœur d'un interprète Scheme, celui de *DrScheme* par exemple, est :

```
(display (eval (read)))
```

La fonction `display` permet d'imprimer une valeur quelconque :

```
;;; display : Valeur -> Rien
;;; (display val) imprime la valeur val
```

La fonction `eval` convertit une S-expression Scheme — représentant un programme — en sa valeur :

```
;;; S-Expression/Programme/ -> Valeur
;;; ERREUR lorsque "prog" ne représente pas un programme Scheme valide
;;; (eval prog) rend la valeur du programme Scheme représenté par la
;;; S-expression "prog".
```

La fonction `read` permet de lire une S-expression quelconque (symbole, nombre, liste, vecteur...). Elle convertit un flux de caractères (provenant d'un clavier, d'un fichier) en une S-expression Scheme :

```
;;; read : -> S-Expression
;;; (read) lit une S-expression tapée au clavier
```

Insistons sur le rôle de cette fonction : il ne faut pas confondre le programme qui est une idée dans la tête du programmeur et ses diverses représentations. Pour un éditeur de texte, c'est une suite de caractères, pour le système d'exploitation un fichier, pour Scheme la valeur produite par `read` c'est-à-dire une S-expression, pour un débogueur des octets liés par des pointeurs. En conclusion, un programme est la description d'un calcul qui peut revêtir différentes représentations suivant les outils dont on dispose et, pour nous écrivain d'un interprète, c'est une S-expression qui nous est fournie par la fonction `read` .

1.2. Spécification de la fonction `deug-eval`

Dans cette partie, nous voudrions (ré)écire la fonction `eval` de l'interprète sous la forme d'une fonction `deug-eval` qui a comme donnée une S-expression — représentant un programme — et qui retourne la valeur de cette S-expression. Ainsi :

```
(deug-eval '(+ 2 3)) → 5
```

et la fonction `deug-eval` a comme spécification :

```
;;; deug-eval: Deug-Programme -> Valeur
;;; (deug-eval p) rend la valeur du programme (de Deug-Scheme) «p».
```

Le langage Scheme que nous analyserons ne sera pas un Scheme complet, ceci pour quatre raisons :

- notre objectif est de voir (tous) les mécanismes fondamentaux et cela ne servirait à rien de traiter un certain nombre de constructions Scheme car elles se traitent comme d'autres constructions que nous avons traitées ;
- nous avons vu en cours et en TD que des constructions de Scheme, très pratiques, ne sont pas indispensables car on peut les remplacer par d'autres (il en est ainsi du `and` , du `or` , du `cond` ...). Lorsque l'on écrit un évaluateur, plus le langage est petit, moins on a de choses à faire.
- c'est ainsi que cela se passe dans la réalité : on commence par ne traiter qu'une partie du langage et, ensuite, on complète l'interpréteur ;
- et enfin, pour une raison technique : un programme Scheme – tel qu'il est défini par la carte de référence – est une suite de définitions et d'expressions, une définition ou une expression étant représentée par une S-expression. Ainsi un programme Scheme est constitué par un certain nombre de S-expressions et la fonction qui évalue un programme Scheme devrait avoir un nombre quelconque d'arguments. Comme nous ne savons pas faire, nous décidons que nos programmes seront réduit à une S-expression.

1.2.1. Cas des programmes « complets »

La restriction précédente n'est pas une restriction fondamentale : si nous avons un programme *P* entier – *i.e.* qui contient des définitions et des expressions — à évaluer, il suffit de l'envelopper dans une forme `(let () P)` avant de

le soumettre à la fonction `deug-eval` . Par exemple, si nous voulons faire évaluer le programme suivant (qui comporte une définition et une expression) :

```
(define  (f n)
   (if (= n 0)
       1
       (* n (f (- n 1)))))   ; fin définition de f
(f 3)
```

nous écrirons :

```
(deug-eval
   '(let ()
       (define  (f n)
          (if (= n 0)
              1
              (* n (f (- n 1)))))   ; fin définition de f
       (f 3)))
```

1.2.2. Auto-évaluation

De plus en plus fort ! `deug-eval` étant écrit en Scheme et évaluant un programme Scheme, on peut le faire évaluer par lui-même. Par exemple, le source de `deug-eval` étant dans la fenêtre de définition et étant compilé, la fenêtre d'interaction peut contenir :

```
(deug-eval  '(let ()
```

```
...
; ; ; deug-eval : Deug-Programme -> Valeur
; ; ; (deug-eval p) rend la valeur du programme
; ; ; (de Deug-Scheme) "p".
(define  (deug-eval  p)
...
```

le source de `deug-eval`

```
   (deug-eval  '(+ 2 3))))
```

Le `deug-eval` de la première ligne est la fonction définie dans la fenêtre de définition (et est donc évalué par DrScheme) alors que le `deug-eval` de la dernière ligne est la fonction définie dans le source de `deug-eval` . Le second `deug-eval` est donc évalué par la définition interne de `deug-eval` qui est elle-même évaluée par le premier `deug-eval` . On parle d'**auto-évaluation**.

Remarque très importante : nous avons dit que nous ne traiterions pas toutes les constructions de Scheme. Mais, si l'on veut pouvoir faire de l'auto-évaluation, il faut que le langage utilisé pour écrire `deug-eval` soit inclus dans le langage que `deug-eval` évalue (en pratique, on s'est arrangé pour que ces deux langages soient égaux).

Remarque non moins importante : dans toute cette partie, nous utiliserons deux langages Scheme : celui de la carte de référence et le langage pour lequel nous écrivons un interprète. Systématiquement (ou tout du moins nous essaierons !), nous nommerons *Scheme* le langage de la carte de référence et *DEUG-Scheme* celui qu'évalue `deug-eval` . Lors de l'auto-évaluation, nous avons même trois langages Scheme :

- celui de DrScheme qui évalue le premier `deug-eval` ,
- celui – c'est un Deug-Scheme – qui est évalué par le premier `deug-eval` (dans l'exemple précédent, tout ce qui est dans la boîte encadrée appartient à ce langage),
- et enfin – c'est aussi un Deug-Scheme, mais pas le même que le précédent puisqu'il n'est pas évalué par la même fonction – celui qui est évalué par le second `deug-eval` (dans l'exemple précédent, `'(+ 2 3)` appartient à ce langage).

et encore, rien ne nous interdit de faire de l'auto-auto-évaluation : on aurait alors quatre niveaux de langages.
Aussi nous parlerons de **Scheme sous-jacent** : le Scheme sous-jacent du Deug-Scheme évalué par le premier `deug-eval` est le Scheme de DrScheme et le Scheme sous-jacent du Deug-Scheme évalué par le second `deug-eval` est le Deug-Scheme évalué par le premier `deug-eval` .

Dernière remarque : l'auto-évaluation peut vous sembler un exercice de style. Il n'en est rien ! ne serait-ce parce qu'elle est excellente pour tester l'évaluateur. En effet, son écriture utilisant toutes les constructions du langage, il les

reste limités, alors qu'il faudrait de nombreux tests « classiques » pour tout tester.

1.3. Grammaire de DEUG-Scheme

Un programme est écrit dans un langage de programmation qui fixe les règles grammaticales que tous les programmes écrits dans ce langage doivent respecter. La syntaxe de bas niveau de Scheme est rudimentaire (des blancs, des parenthèses et des mots sans oublier quelques règles d'abréviations concernant l'apostrophe). La grammaire de Scheme comporte des formes spéciales, des applications fonctionnelles, des variables et des citations. Ce faisant on plaque une interprétation sur les S-expressions qui sont lues.

La grammaire du sous-ensemble de Scheme utilisé dans ce cours est indiqué en tête du listing donné en annexe (page 28).

- Comme déjà indiqué, un programme *DEUG-Scheme* est simplement une expression *DEUG-Scheme* (il n'y en a qu'une et il n'y a pas de définition au top-level).
- Il n'y a pas de `and` , de `or` et de `let *`, ces constructions n'étant pas essentielles.
- En revanche, nous avons conservé le `cond` , qui n'est pas essentiel non plus, mais qui est tellement pratique pour écrire les sources de `deug-eval` (et nous l'avons donc mis dans le langage en vue de l'auto-évaluation).

1.4. Structure générale du listing

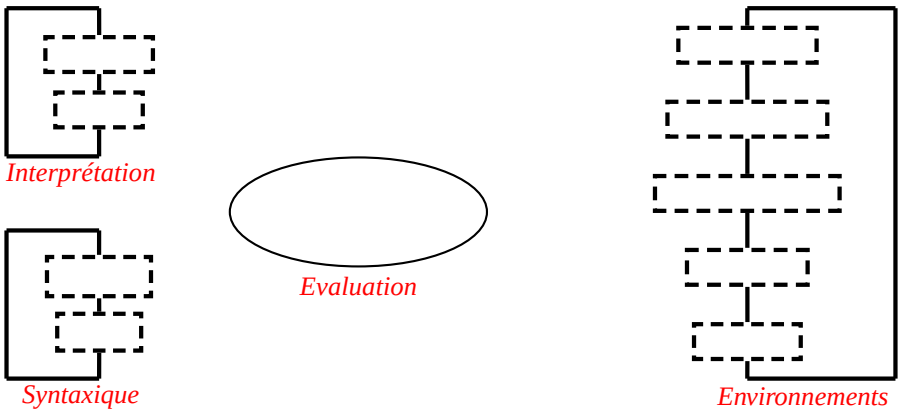
Tout d'abord, comme nous l'avons déjà dit, le listing contient (bien sûr sous forme de commentaires) la grammaire du langage utilisé :

```
5   ;;;; {{{ Grammaire du langage
32  ;;;; }}} Grammaire du langage
```

On trouve ensuite le source proprement dit. Dans un premier temps, nous définissons des fonctions de service utiles pour écrire le programme :

```
34  ;;;; {{{ Utilitaires généraux
100 ;;;; }}} Utilitaires généraux
```

Pour le programme proprement dit, comme pour les expressions booléennes que nous avons vues dans les cours précédents, nous écrirons une barrière syntaxique, une barrière d'interprétation et une barrière d'abstraction des environnements :



Il est facile de déterminer la barrière syntaxique (il suffit de regarder la grammaire), et nous le ferons tout de suite.

En revanche, pour la barrière d'évaluation et la barrière d'abstraction des environnements, il n'est pas aisé de trouver *a priori* les fonctions nécessaires, aussi nous reporterons cette étude plus tard.

Ainsi, nous donnerons d'abord la barrière syntaxique :

```
102 ;;;; {{{ Barrière-syntaxique
224 ;;;; }}} Barrière-syntaxique
```

premiers permettra d'écrire l'évaluateur (la fonction deug-eval) :

```
226  ;;;; {{{ Evaluateur
350  ;;;; }}} Evaluateur
```

au dessus d'une barrière d'interprétation :

```
352  ;;;; {{{ Barrière-interpretation
475  ;;;; }}} Barrière-interpretation
```

et d'une barrière d'abstraction des environnements :

```
477  ;;;; {{{ Environnements-H (barrière de haut niveau)
677  ;;;; }}} Environnements-H (barrière de haut niveau)
```

Enfi n, nous avons écrit, bien sûr sous forme de commentaires, le mode d'emploi de notre logiciel :

```
679  ;;;; {{{ Mode d'emploi
697  ;;;; }}} Mode d'emploi
```

2. Utilitaires généraux

Dans un premier temps, nous défi nissons des fonctions utilitaires classiques (`cadr` , `cddr` ... `length` , `member` ... qui ont presque toutes été vues dans le présent ouvrage, celles qui n'ont pas été vues étant très faciles. Deux fonctions, `deug-erreur` et `deug-map` , demandent tout de même une explication.

Fonction `deug-erreur`

Nous avons défi ni la fonction `deug-erreur` pour personaliser les messages d'erreurs de `deug-eval` (ils commencent tous par *deug-eval*) :

```
39 ;;; Signaler une erreur et abandonner l'évaluation.
   (define (deug-erreur fn message donnee)
     (erreur 'deug-eval fn message donnee) )
```

Fonction `deug-map`

Nous avons défi ni une fonction `deug-map` (et non `map`) car, en Scheme, la fonction `map` opère sur un nombre quelconque de listes, la fonction appliquée étant une fonction n-aire (alors que nous l'avons toujours utilisée – et nous l'utiliserons encore ainsi dans le présent logiciel – avec une fonction unaire et une seule liste) :

```
75 ;;; deug-map: (alpha -> beta) * LISTE[alpha] -> LISTE[beta]
   ;;; (deug-map f L) rend la liste des valeurs de «f» appliquée aux termes
   ;;; de la liste «L».
   (define (deug-map f L)
     (if (pair? L)
         (cons (f (car L)) (deug-map f (cdr L)))
         '() ) )
```

3. Barrière syntaxique

La barrière syntaxique est constituée d'une longue kyrielle de fonctions pour reconnaître (reconnaisseurs qui ont pour type de résultat `bool`) et extraire (accesseurs) de l'information des diverses formes syntaxiques possibles. Contrairement à ce que nous avons fait jusqu'alors, elles sont données règle de grammaire par règle de grammaire et non en écrivant d'abord les reconnaisseurs puis les accesseurs.

Règle `expression := ...` *(début)*

```
108  ;;; variable ? : Expression -> bool
124  ;;; citation ? : Expression -> bool
133  ;;; conditionnelle ? : Expression -> bool
137  ;;; conditionnelle-clauses : Conditionnelle -> LISTE[Clause]
141  ;;; alternative ? : Expression -> bool
145  ;;; alternative-condition : Alternative -> Expression
149  ;;; alternative-consequence : Alternative -> Expression
153  ;;; alternative-alternant : Alternative -> Expression
159  ;;; sequence ? : Expression -> bool
163  ;;; sequence-exps : Sequence -> LISTE[Expression]
```

Noter, encore une fois, comme ces spécifications suivent la grammaire. Pour l'implantation, rien à dire, hormis peut-être les implantations un peu longues de `variables?` (pour exclure les mots-clefs) et `citation?` (car nous avons mis ensemble les constantes et les données « quotées »).

Noter la définition de `alternative-alternant` : l'alternant est facultatif. Lorsque la condition est fausse et que l'alternant est absent, la fonction `alternative-alternant` retourne false, ce qui est permis par la norme de Scheme.

Règle `expression` := ... *(suite et fin)*

```
167  ;;; bloc ? : Expression -> bool
171  ;;; bloc-liaisons : Bloc -> LISTE[Liaison]
175  ;;; bloc-corps : Bloc -> Corps
179  ;;; application ? : Expression -> bool
183  ;;; application-fonction : Application -> Expression
187  ;;; application-arguments : Application -> LISTE[Expression]
```

On suit toujours la règle de grammaire...

Règle `clause` := ...

```
191  ;;; clause-condition : Clause -> Expression
195  ;;; clause-expressions : Clause -> LISTE[Expression]
```

Rien à dire...

Règle `liaison` := ...

```
199  ;;; liaison-variable : Liaison -> Variable
203  ;;; liaison-exp : Liaison -> Expression
```

Rien à dire...

Règle `corps` := ...

```
207  ;;; definition ? : Corps -> bool
     ;;; (definition ? corps) rend #t ssi le premier élément du corps
     ;;; "corps" est une définition
```

Ici, nous avons donné la spécification complète car c'est une forme de règle de grammaire que nous n'avons encore pas vue : un corps est une suite (éventuellement vide) de définitions suivie d'une suite (non vide) d'expressions. Deux applications de cette règle diffèrent donc – et c'est pourquoi nous donnons un reconnaisseur – selon que le premier élément est, ou n'est pas, une définition :

Règle `definition` := ...

```
213  ;;; definition-nom-fonction : Definition -> Variable
217  ;;; definition-variables : Definition -> LISTE[Variable]
221  ;;; definition-corps : Definition -> Corps
```

Facile...

4. Implantation de l'évaluateur

4.1. Implantation de la fonction `deug-eval`

Comme dans le cas des expressions booléennes, on évalue une expression dans un environnement où des variables (nommant des fonctions ou des données) sont liées à leurs valeurs.

Lorsque vous avez écrit des programmes Scheme en TP, vous avez utilisé des fonctions comme `+`, `-`, `car` ... Vous étiez donc dans un certain environnement, l'environnement initial.

Ainsi la définition de `deug-eval` est :

```
229 ;;; deug-eval: Deug-Programme -> Valeur
    ;;; (deug-eval p) rend la valeur du programme (de Deug-Scheme) «p».
    (define (deug-eval p)
      (evaluation p (env-initial) ) )
```

avec :

```
234 ;;; evaluation: Expression * Environnement -> Valeur
    ;;; (evaluation exp env) rend la valeur de l'expression «exp» dans
    ;;; l'environnement «env».
624 ;;; env-initial: -> Environnement
    ;;; (env-initial) rend l'environnement initial, i.e. l'environnement qui
    ;;; contient toutes les primitives.
```

4.2. Implantation de la fonction `evaluation`

La définition de la fonction `evaluation` n'est qu'un aiguillage qui analyse la nature de l'expression à évaluer et appelle une fonction ad-hoc, avec comme arguments le contenu de la forme syntaxique (et non l'expression à analyser) :

```
237 (define (evaluation exp env)
      ;; (discrimine l'expression et invoque l'évaluateur spécialisé)
      (cond
       ((variable? exp)          (variable-val exp env))
       ((citation? exp)          (citation-val exp))
       ((alternative? exp)       (alternative-eval
                                  (alternative-condition exp)
                                  (alternative-consequence exp)
                                  (alternative-alternant exp) env))
       ((conditionnelle? exp)    (conditionnelle-eval
                                  (conditionnelle-clauses exp) env))
       ((sequence? exp)          (sequence-eval (sequence-exps exp) env))
       ((bloc? exp)              (bloc-eval (bloc-liaisons exp)
                                            (bloc-corps exp) env))
       ((application? exp)       (application-eval
                                  (application-fonction exp)
                                  (application-arguments exp) env))
       (else (deug-erreur 'evaluation "pas un programme" exp)) )
```

Ces différentes fonctions sont des évaluateurs spécialisés, dont nous étudions les définitions dans les paragraphes qui suivent, sauf

- la fonction `variable-val` qui devra être une des fonctions de la barrière des environnements,
- la fonction `citation-val` qui devra être une des fonctions de la barrière d'interprétation.

4.2.1. Implantation de la fonction `alternative-eval`

La définition de cette fonction ne devrait pas vous poser de problème :

```
236 ;;; alternative-eval: Expression^3 * Environnement -> Valeur
    ;;; (alternative-eval condition consequence alternant env) rend la valeur de
    ;;; l'expression «(if condition consequence alternant)» dans l'environnement «env».
    (define (alternative-eval condition consequence alternant env)
      (if (evaluation condition env)
          (evaluation consequence env)
          (evaluation alternant env)) )
```

4.2.2. Implantation de la fonction `conditionnelle-eval`

Nous avons dit que la conditionnelle n'était pas essentielle (toute conditionnelle peut être réécrite avec des alternatives), mais que nous la mettions tout de même dans DEUG-Scheme car elle est très pratique (nous nous en sommes déjà servi dans la définition de `evaluation`).

Dans DEUG-Scheme, c'est la seule forme non essentielle. Les formes `and` et `or` ont été directement réécrites dans la définition de la fonction `deug-eval` plutôt que par programme (dans la première version de `deug-eval` , il y n'en y avait que quelques unes, dans les reconnaisseurs syntaxiques surtout).

En fait, dans les vrais évaluateurs, existe une phase dite de macro-expansion qui réduit le nombre de constructions qu'emploie un programme à celles qui sont vraiment essentielles et c'est ce que nous allons faire pour la conditionnelle :

```
264 ;;; LISTE[Clause] * Environnement -> Valeur
    ;;; (conditionnelle-eval clauses env) rend la valeur, dans l'environnement «env»,
    ;;; de l'expression «(cond c1 c2 ... cn)», «c1», «c2»... «cn» étant les éléments
    ;;; de la liste «clauses».
    (define (conditionnelle-eval clauses env)
      (evaluation (conditionnelle-expansion clauses) env) )
```

la fonction `conditionnelle-expansion` ayant comme spécification :

```
271 ;;; conditionnelle-expansion: LISTE[Clause] -> Expression
    ;;; (conditionnelle-expansion clauses) rend l'expression, écrite avec des
    ;;; alternatives, équivalente à l'expression «(cond c1 c2 ... cn)»,
    ;;; «c1», «c2»... «cn» étant les éléments de la liste «clauses».
```

Nous n'étudierons pas l'implantation de cette fonction car vous l'avez vue en TD.

4.2.3. Implantation de la fonction `sequence-eval`

```
289 ;;; sequence-eval: LISTE[Expression] * Environnement -> Valeur
    ;;; (sequence-eval exps env) rend la valeur, dans l'environnement «env», de
    ;;; l'expression «(begin e1 ... en)», «e1»... «en» étant les éléments de la liste
    ;;;  «exps».
    ;;; (Il faut évaluer tour à tour les expressions et rendre la valeur de la
    ;;;  dernière d'entre elles.)
```

Tout d'abord notons qu'une séquence peut être vide (i.e. l'expression `(begin)` est correcte). La norme ne précise pas ce que l'on doit rendre dans ce cas ; nous avons choisi de rendre `#f` .

Rappelons que pour évaluer `(begin e1 e2 ... en)` , il faut évaluer tour à tour les expressions `e1` , `e2` ... `en` et rendre la valeur de cette dernière. Ainsi la dernière expression doit être traitée de façon différente. Pour ce faire, pour l'efficacité, nous avons déjà vu sur d'autres exemples qu'il fallait définir une fonction interne, de même spécification que la fonction principale, mais dont la donnée est une liste non vide (et nous en profitons pour globaliser la varaible `env`) :

```
295 (define (sequence-eval exps env)
      ;; sequence-eval+ : LISTE[Expression]/non vide/ -> Valeur
      ;; même fonction, sachant que la liste "exps" n'est pas vide et en globalisant la variable "env".
      (define (sequence-eval+ exps)
        ... à faire
        ... à faire
        ... à faire
        ... à faire                                                    )
      ;; expression de (sequence-eval exps env) :
      (if (pair? exps)
          (sequence-eval+ exps)
          #f ) )
```

Pour implanter la fonction `sequence-eval+`, deux cas selon qu'il n'y a qu'une expression (c'est alors la dernière) ou plusieurs :

```
295 (define (sequence-eval exps env)
      ;; sequence-eval+ : LISTE[Expression]/non vide/ -> Valeur
      ;; même fonction, sachant que la liste "exps" n'est pas vide et en globalisant la variable "env".
      (define (sequence-eval+ exps)
        (if (pair? (cdr exps))
            ... à faire
            ... à faire
            ... à faire                                                )
      ;; expression de (sequence-eval exps env) :
      (if (pair? exps)
          (sequence-eval+ exps)
          #f ) )
```

s'il n'y en a qu'une, il suffit de l'évaluer (et de rendre sa valeur) :

```
295 (define (sequence-eval exps env)
      ;; sequence-eval+ : LISTE[Expression]/non vide/ -> Valeur
      ;; même fonction, sachant que la liste "exps" n'est pas vide et en globalisant la variable "env".
      (define (sequence-eval+ exps)
        (if (pair? (cdr exps))
            ... à faire
            ... à faire
            (evaluation (car exps) env)))
      ;; expression de (sequence-eval exps env) :
      (if (pair? exps)
          (sequence-eval+ exps)
          #f ) )
```

s'il y en a plusieurs, on évalue la première sans se soucier de sa valeur puis on évalue la séquence des expressions qui restent (le tout à l'aide d'une séquence) :

```
295 (define (sequence-eval exps env)
      ;; sequence-eval+ : LISTE[Expression]/non vide/ -> Valeur
      ;; même fonction, sachant que la liste "exps" n'est pas vide et en globalisant la variable "env".
      (define (sequence-eval+ exps)
        (if (pair? (cdr exps))
            (begin (evaluation (car exps) env)
                   (sequence-eval+ (cdr exps)) )
            (evaluation (car exps) env)))
      ;; expression de (sequence-eval exps env) :
      (if (pair? exps)
          (sequence-eval+ exps)
          #f ) )
```

4.2.4. Implantation de la fonction `application-eval`

```
309 ;;; application-eval: Expression * LISTE[Expression] * Environnement -> Valeur
    ;;; (application-eval exp-fn arguments env) rend la valeur de l'invocation de
    ;;; l'expression «exp-fn» aux arguments «arguments» dans l'environnement «env».
```

Notons que `exp-fn` est une expression dont la valeur doit être fonctionnelle et que `arguments` est une liste d'expressions. Ainsi, on doit :

- évaluer la valeur de la fonction,
- évaluer les valeurs des arguments ; ceux-ci sont représentés par une liste, on pense donc à un map (rappelons que nous avons nommé `deug-map` cette fonction), mais ce n'est pas aussi simple car la fonction `evaluation` a deux arguments, l'expression mais aussi l'environnement : comme nous l'avons déjà fait dans ce cas, nous définirons une fonction interne,
- appliquer (on parlera ici d'invocation) la valeur de la fonction aux valeurs des arguments.

De plus, nous voudrions vérifier que la valeur de l'expression `exp-fn` est bien une valeur fonctionnelle.
D'où la définition :

```
312 (define (application-eval exp-fn arguments env)
      ;; eval-env : Expression -> Valeur
      ;; (eval-env exp) rend la valeur de «exp» dans l'environnement «env»
      (define (eval-env exp)
        (evaluation exp env))
      ;; expression de (application-eval exp-fn arguments env) :
      (let ((f (evaluation exp-fn env)))
        (if (invocable? f)
            (invocation f (deug-map eval-env arguments))
            (deug-erreur 'application-eval
                         "pas une fonction" f ) ) ) )
```

Les fonctions `invocable?` et `invocation` doivent faire partie de la barrière d'interprétation :

```
;;; invocable?: Valeur -> bool
;;; invocation: Invocable * LISTE[Valeur] -> Valeur
```

- `(invocable? val)` rendant vrai si, et seulement si, `val` est une fonction (primitive ou définie par le programmeur),
- `(invocation f vals)` rendant la valeur de l'application de `f` aux éléments de `vals`.

4.2.5. Implantation de la fonction `bloc-eval`

```
324 ;;; bloc-eval: LISTE[Liaison] * Corps * Environnement -> Valeur
    ;;; (bloc-eval liaisons corps env) rend la valeur, dans l'environnement «env»,
    ;;; de l'expression «(let liaisons corps)».
```

Rappelons la sémantique du `let` : nous devons évaluer le corps dans un environnement obtenu en ajoutant les liaisons à l'environnement courant. D'où la définition :

```
327 (define (bloc-eval liaisons corps env)
      (corps-eval corps (env-add-liaisons liaisons env)) )
```

la fonction `env-add-liaisons` devant être une fonction de la barrière des environnements :

```
;;; env-add-liaisons: LISTE[Liaison] * Environnement -> Environnement
```

`(env-add-liaisons liaisons env)` rendant l'environnement obtenu en ajoutant, à l'environnement `env`, les liaisons `liaisons`.

4.2.6. Implantation de la fonction `corps-eval`

```
330 ;;; corps-eval: Corps * Environnement -> Valeur
    ;;; (corps-eval corps env) rend la valeur de «corps» dans l'environnement «env»
```

Tout d'abord des rappels :

- un corps est une suite de définitions et d'expressions,
  - les définitions sont toutes écrites avant les expressions,

– il peut ne pas y avoir de définitions,
– il y a obligatoirement une expression,
– en fait les expressions constituent une séquence (implicite) ;

– pour évaluer un corps, on évalue la séquence du corps dans l'environnement obtenu en enrichissant l'environnement courant avec les définitions du corps.

D'où la définition :

```
(define (corps-eval corps env)
  (let ((def-exp (corps-separation-defs-exps corps)))
    (let ((defs (car def-exp))
          (exp (cadr def-exp)))
      (evaluation exp (env-enrichissement env defs)) ) ) )
```

la fonction `corps-separation-defs-exps` permettant de séparer les définitions et les expressions présentes dans le corps. Plus précisément :

```
;;; corps-separation-defs-exps: Corps -> (LISTE[Definition] * LISTE[Expression])
;;; (corps-separation-defs-exps corps) rend une liste dont le premier élément est
;;; la liste des definitions du corps «corps» et les autres éléments sont les
;;; expressions de ce corps.
```

et la fonction `env-enrichissement` devant être une fonction de la barrière des environnements :

Dans barrière des environnements :

```
;;; env-enrichissement: Environnement * LISTE[Definition] -> Environnement
```

`(env-enrichissement env defs)` rendant l'environnement `env` étendu avec un bloc d'activation pour les définitions fonctionnelles `defs` .

Implantation de la fonction `corps-separation-defs-exps`

```
;;; corps-separation-defs-exps: Corps -> (LISTE[Definition] * LISTE[Expression])
;;; (corps-separation-defs-exps corps) rend une liste dont le premier élément est
;;; la liste des definitions du corps «corps» et les autres éléments sont les
;;; expressions de ce corps.
```

Noter bien le type du résultat de cette fonction : c'est une liste (hétérogène), le premier élément étant une liste de définitions et les autres éléments étant des expressions (les expressions du corps).

L'idée de la définition est simple :

– si le premier élément du corps est une définition, nous devons la rajouter dans la liste des définitions du corps privé de cette définition (d'où un appel récursif),
– si le premier élément n'est pas une définition, la liste des définitions est vide, liste vide que nous devons mettre devant la liste des expressions :

```
(define (corps-separation-defs-exps corps)
  (if (definition? (car corps))
      (let ((def-exp-cdr
              (corps-separation-defs-exps (cdr corps))))
        (cons (cons (car corps)
                    (car def-exp-cdr))
              (cdr def-exp-cdr)))
      (cons '() corps) ) )
```

Et nous avons fini d'écrire l'évaluateur, il ne nous reste plus qu'à implanter la barrière d'interprétation et la barrière des environnements.

5. Barrière d'interprétation

5.1. Introduction

Nous avons dit que nous devions manipuler deux sortes de valeurs, les valeurs non fonctionnemlles (les entiers, les booléens... les listes...) et les valeurs fonctionnelles et, dans l'évaluateur, nous avons utilisé des fonctions différentes pour ces deux types de valeurs. Nous retrouvons donc ces deux sortes de valeurs dans la barrière d'interprétation :

```
352 ;;;;   {{{  Barrière-interpretation
357 ;;;;        {{{  Valeurs-non-fonctionnelles
366 ;;;;        }}}  Valeurs-non-fonctionnelles
368 ;;;;        {{{  Valeurs-fonctionnelles
474 ;;;;        }}}  Valeurs-fonctionnelles
475 ;;;;   }}}  Barrière-interpretation
```

5.2. Valeurs non fonctionnelles

Une seule fonction a été utile dans l'écriture de l'évaluateur :

```
360 ;;; citation-val: Citation -> Valeur/non fonctionelle/
    ;;; (citation-val cit) rend la valeur de la citation «cit».
```

Nous décidons d'implanter les valeurs non fonctionnelles par leur valeur dans le Scheme sous-jacent. La défi nition est alors très simple :

```
362 (define   (citation-val    cit)
      (if  (pair?   cit)
           (cadr   cit)
           cit  ) )
```

5.3. Valeurs fonctionnelles

5.3.1. Spécification de la barrière d'abstraction

Tout d'abord, récapitulons les fonctions (Scheme) que nous avons utilisées dans l'évaluateur pour manipuler les fonctions (de Deug-Scheme) :

```
;;; invocable?: Valeur -> bool
;;; (invocable? val) rend vrai ssi «val» est une fonction (primitive ou définie
;;; par le programmeur)
;;; invocation: Invocable * LISTE[Valeur] -> Valeur
;;; (invocation f vals) rend la valeur de l'application de «f» aux éléments de
;;; «vals».
```

Noter que nous devrons aussi créer des valeurs fonctionnelles (pour enrichir l'environnement et pour défi nir l'environnement initial).

5.3.2. Implantation de la barrière d'abstraction

En fait, les fonctions sont de deux natures très différentes, celles-ci pouvant être :

1. les fonctions défi nies (dans Deug-Scheme) par le programmeur,
2. les fonctions primitives, c'est-à-dire
   - celles qui sont fournies par le langage (comme +, car..),
   - que nous utilisons (ou non, mais toutes celles qui sont utilisées doivent être prédéfi nies si l'on veut pouvoir autoévaluer l'interprèteur) dans l'écriture de l'interprète,
   - qui, dans un interprète du commerce, sont codées en langage machine.

   Bien sûr, nous ne les coderons pas dans le langage machine, nous les coderons tout simplement en utilisant une fonction du Scheme sous-jacent, c'est-à-dire le Scheme dans lequel on écrit l'évaluateur (par exemple, DrScheme ou, lors de l'auto-évaluation, Deug-Scheme).

Aussi nous décidons d'implanter la barrière d'abstraction des valeurs fonctionnelles en créant, et utilisant, une barrière d'abstraction des primitives et une barrière d'abstraction des fonctions défi nies par le programmeur. Ainsi la structure de la partie du listing qui met en œuvre la barrière d'interprétation pour les valeurs fonctionnelles est :

```
388 ;;;;          {{{  Valeurs-fonctionnelles
    définition de invocable?
    définition de invocation
390 ;;;;               {{{  Primitives
437 ;;;;               }}}  Primitives
439 ;;;;               {{{  Fonctions-definies
473 ;;;;               }}}  Fonctions-definies
474 ;;;;          }}}  Valeurs-fonctionnelles
```

Implantation de `invocable?`

```
374 ;;; invocable?: Valeur -> bool
    ;;; (invocable? val) rend vrai ssi «val» est une fonction (primitive ou définie
    ;;; par le programmeur)
    (define (invocable? val)
      (if (primitive? val)
          #t
          (fonction? val) ) )
```

Remarque : dans la première version de `deug-eval` , nous avions écrit :

```
(define (invocable? val)
  (or (primitive? val) (fonction? val) ) )
```

Nous avons transformé la défi nition pour ne pas avoir besoin de la forme `or` .

Implantation de `invocation`

```
382 ;;; invocation: Invocable * LISTE[Valeur] -> Valeur
    ;;; (invocation f vals) rend la valeur de l'application de «f» aux éléments de
    ;;; «vals».
    (define (invocation f vals)
      (if (primitive? f)
          (primitive-invocation f vals)
          (fonction-invocation f vals) ) )
```

5.3.3. Barrière d'abstraction des primitives

Structure de données

Nous décidons d'implanter une primitive par un 4-uplet :

- le premier élément est le symbole *primitive* (pour reconnaître les primitives parmi les listes),
- le second élément est la fonction du Scheme sous-jacent qui implante la primitive,
- le troisième élément est un comparateur (= ou >=),
- le dernier élément est un entier naturel, ces deux derniers éléments permettant de spécifi er l'arité de la primitive. Par exemple,
  - pour une primitive d'arité 2, le troisième élément est = et le quatrième élément est 2,
  - pour une primitive qui peut avoir 1, 2, 3... arguments, le troisième élément est >= et le quatrième élément est 1.

Par exemple, la primitive correspondant à `'+` est `(list '*primitive * + >= 1)`.

Implantation de `primitive?`

Pour savoir si une valeur est une primitive, il suffi t de vérifi er que c'est une liste ayant au moins un élément dont le premier élément est le symbole `*primitive *` :

```
400 ;;; primitive?: Valeur -> bool
    ;;; (primitive? val) rend vrai ssi «val» est une fonction primitive.
    (define (primitive? val)
      (if (pair? val)
          (equal? (car val) '*primitive *)
          #f) )
```

Implantation de `primitive-creation`

```
;;; primitive-creation: N-UPLET[(Valeur -> Valeur)*(num * num -> bool)*num]
;;;                 -> Primitive
;;; (primitive-creation f-c-n) rend la primitive implantée par la fonction (du
;;; Scheme sous-jacent) «f», le premier élément de «f-c-n», et dont l'arité est
;;; spécifiée par le comparateur «c», deuxième élément de «f-c-n» et l'entier «n»,
;;;  troisième élément de «f-c-n».
(define (primitive-creation f-c-n)
  (cons '*primitive * f-c-n) )
```

Spécification de `primitive-invocation`

La définition de `primitive-invocation` est un peu plus compliquée. Voyons tout d'abord sa spécification :

```
;;; primitive-invocation: Primitive * LISTE[Valeur] -> Valeur
;;; (primitive-invocation p vals) rend la valeur de l'application de la
;;; primitive «p» aux éléments de «vals».
```

Idée pour l'implantation de `primitive-invocation`

Raisonnons sur un exemple en considérant l'évaluation (par DrScheme) de `(deug-eval '(+ 2 3))` :

`(deug-eval '(+ 2 3))` ⇒ ...

en nommant `env-ini` la valeur de l'environnement initial, en « faisant tourner à la main » notre évaluateur, au bout de quelques pas on obtient (comme exercice, vous pouvez vérifier cette assertion) :

```
... ⇒   (primitive-invocation (variable-val '+ env-ini)
                              '(2 3) env-ini)
```

et comme nous avons dit que la primitive correspondant à `'+` était implantée par la liste `(list '*primitive * + >= 1)` :

```
... ⇒   (primitive-invocation (list '*primitive * + >= 1)
                              '(2 3) env-ini)
```

et nous voulons obtenir

```
... ⇒   (+ 2 3)
```

Comment, à partir de `(list '*primitive * + >= 1)` (qui est la valeur de l'argument `primitive`) et `'(2 3)` (qui est la valeur de l'argument `vals`) obtenir `(+ 2 3)` ?

- pour retrouver la fonction +, c'est facile puisque c'est le second élément de la liste `(list '*primitive * + >= 1)` ,
- pour obtenir 2 et 3 remarquons que nous ne pouvons pas écrire, comme corps de `primitive-invocation` , `((cadr primitive) vals)` puisque, sur l'exemple, cela donnerait `(+ '(2 3))` qui n'est pas le résultat recherché. Si on veut le faire, à la main, sur notre exemple, c'est facile : 2 (resp. 3) est le premier (resp. deuxième) élément de la liste `'(2 3)` et le calcul de `(primitive-invocation (list '*primitive * + >= 1) '(2 3) env-ini)` doit être :

```
... ⇒   ((cadr (list '*primitive * + >= 1)) (car '(2 3))
                                            (cadr '(2 3)))

... ⇒   (+ 2 3)
```

Mais, si cette extraction est facile au coup par coup et à la main, elle est plus délicate lorsque l'on veut écrire un programme qui l'effectue en général. En effet l'extraction dépend du nombre d'arguments de la primitive, autrement dit de la longueur de la liste `vals` .

Implantation de primitive-invocation

De plus, nous voudrions vérifier (une fois n'est pas coutume) que la primitive est invoquée avec un nombre d'argument correct.

```
419 (define (primitive-invocation primitive vals)
      (let ((n (length vals))
            (f (cadr primitive))
            (compare (caddr primitive))
            (arite (cadddr primitive)))
        (if (compare n arite)
            (cond
             ((= n 0) (f))
             ((= n 1) (f (car vals)))
             ((= n 2) (f (car vals) (cadr vals)))
             ((= n 3) (f (car vals) (cadr vals) (caddr vals)))
             ((= n 4) (f (car vals) (cadr vals)
                         (caddr vals) (cadddr vals) ))
             (else
              (deug-erreur 'primitive-invocation
                           "limite implantation (arités quelconques < 5)"
                           vals)))
            (deug-erreur 'primitive-invocation "arité incorrecte" vals) ) ) )
```

- Tout d'abord, nous nommons le nombre d'arguments,
- ainsi que les informations issues de la représentation de la primitive : la fonction du Scheme sous-jacent, l'opération de comparaison et la valeur de l'arité ;
- après avoir vérifié que l'arité était correcte (en affichant éventuellement un message d'erreur), il ne reste plus qu'à appliquer la fonction du Scheme sous-jacent aux arguments et, pour ce faire, nous faisons un `cond` sur le nombre d'arguments.

Noter que, de part l'implantation, nous limitons les arités quelconques (à 4) ce qui est fait, dans certains interprètes Scheme (mais, bien sûr avec une valeur plus grande).

5.3.4. Barrière d'abstraction des fonctions définies par le programmeur

Pour déterminer la structure de données que nous utilisons pour les fonctions définies par le programmeur, considérons un « exemple » d'une définition de fonction :

```
(define (f v1 v2)
  corps)
```

et d'une d'application de cette fonction (`e1` et `e2` étant des expressions) :

```
(f e1 e2)
```

Lorsque nous avons étudié la sémantique des applications de fonctions, nous avons dit que l'on évaluait (par substitution) les arguments et que, ensuite, on remplaçait l'appel de fonction par le corps de la fonction en substituant aux variables les valeurs des arguments correspondant, la valeur des autres variables – fonctionnelles ou non fonctionnelles – étant pris dans l'environnement où est définie la fonction.

Dans l'implantation que nous avons donné de `deug-eval`, l'évaluateur spécialisé `application-eval` évalue les différents arguments. Dans notre exemple, supposons que `e1` est évalué en `a1` et `e2` est évalué en `a2`.

```
(f e1 e2)   ⇒   (f a1 a2)
```

Pour évaluer l'application `(f e1 e2)`, il ne reste donc plus qu'à évaluer le corps – de la définition de la fonction – en substituant aux variables – de la définition de la fonction – (`e1` et `e2`) les valeurs des arguments correspondant (`a1` et `a2`), dans l'environnement où est définie la fonction. Autrement dit, en terme d'environnement, on doit évaluer le corps de la définition dans l'environnement obtenu en ajoutant à l'environnement où est définie la fonction les liaisons `v1 -> a1` et `v2 -> a2` :

Structure de données

En résumé, pour pouvoir ensuite évaluer des applications de la fonction, nous devons connaître la liste des variables et le corps de sa définition ainsi que l'environnement où est définie la fonction. Nous décidons alors d'implanter les

fonctions définies par le programmeur par un 4-uplet :

- le premier élément est le symbole `*fonction*` (pour reconnaître les fonctions définies par le programmeur parmi les listes),
- le second élément est la liste des variables de la définition de la fonction,
- le troisième élément est le corps de la définition de la fonction,
- le quatrième élément est l'environnement où est définie la fonction.

Implantation de `fonction?`

Pour savoir si une valeur est une fonction définie par le programmeur, il suffit de vérifier que c'est une liste ayant au moins un élément dont le premier élément est le symbole `*fonction*` :

```
448 ;;; fonction?: Valeur -> bool
    ;;; (fonction? val) rend vrai ssi «val» est une fonction créée par le programmeur.
    (define (fonction? val)
      (if (pair? val)
          (equal? (car val) '*fonction*)
          #f ) )
```

Implantation de `fonction-creation`

L'implantation de `fonction-creation` va de soi :

```
464 ;;; fonction-creation: Definition * Environnement -> Fonction
    ;;; (fonction-creation definition env) rend la fonction définie par
    ;;; «definition» dans l'environnement «env».
    (define (fonction-creation definition env)
      (list '*fonction*
            (definition-variables definition)
            (definition-corps definition)
            env ) )
```

Implantation de `fonction-invocation`

Comme nous l'avons dit, pour évaluer une application de la fonction, il suffit d'évaluer le corps de la définition – qui est le troisième élément du 4-uplet qui mémorise la fonction – en étendant l'environnement – quatrième élément du 4-uplet – en liant les variables – second élément du 4-uplet – avec les arguments. D'où la définition :

```
455 ;;; fonction-invocation: Fonction * LISTE[Valeur] -> Valeur
    ;;; (fonction-invocation f vals) rend la valeur de l'application de
    ;;; la fonction définie par le programmeur «f» aux éléments de «vals».
    (define (fonction-invocation f vals)
      (let ((variables (cadr f))
            (corps (caddr f))
            (env (cadddr f)) )
        (corps-eval corps (env-extension env variables vals)) ) )
```

la fonction `env-extension` devant être une fonction de base des environnements avec comme spécification :

Dans barrière des environnements :

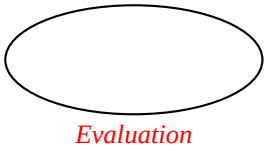
```
;;; env-extension: Environnement * LISTE[Variable] * LISTE[Valeur] -> Environnement
;;; (env-extension env vars vals) rend l'environnement «env» étendu avec
;;; un bloc d'activation liant les variables «vars» aux valeurs «vals».
```

Et l'implantation de la barrière d'évaluation est terminée.

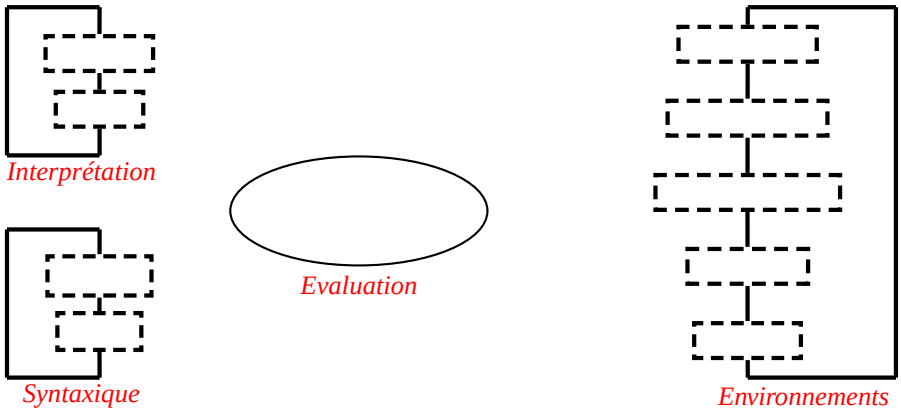
6. Barrière d'abstraction des environnements

6.1. État des lieux

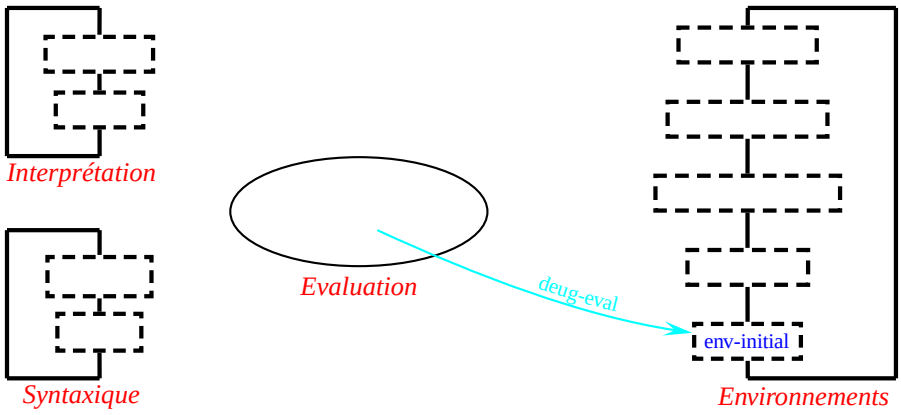
Avant d'étudier la barrière d'abstraction des environnements, nous voudrions schématiser la structure de l'évaluateur (en oubliant les fonctions de service). Nous voulions donc écrire un évaluateur :



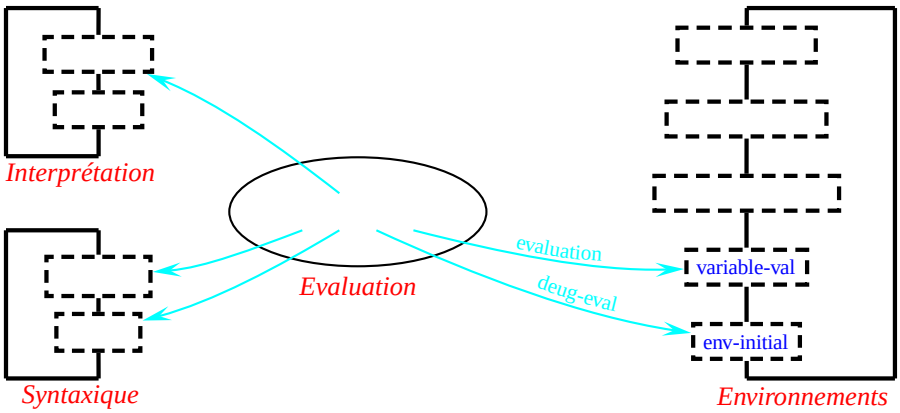
en sachant, dès le départ, qu'il faudrait une barrière syntaxique, que nous avons défi nie en premier, une barrière d'interprétation que nous venons de défi nir, et une barrière des environnements :



Après avoir défi ni la barrière syntaxique, nous avons défi ni la fonction `deug-eval` en utilisant la fonction `evaluation` et la fonction – de la barrière des environnements – `env-initial` :

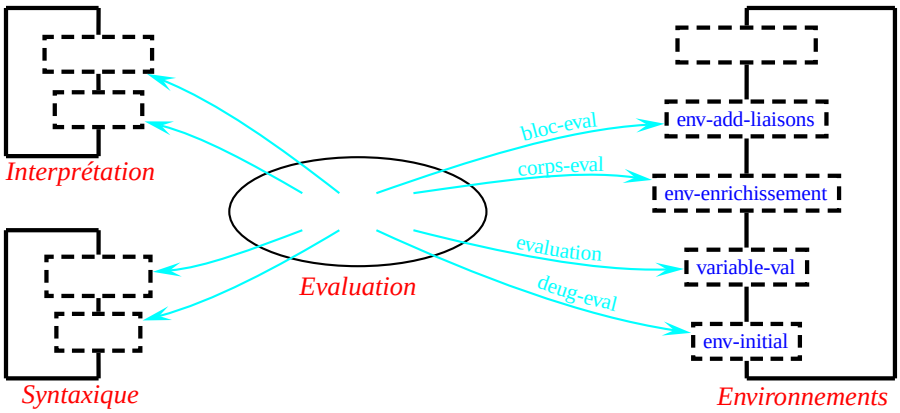


La défi nition de la fonction `evaluation` utilise les fonctions de la barrière d'abstraction, des évaluateurs spécialisés, la fonction – de la barrière d'interprétation – `citation-val` et la fonction – de la barrière des environnements – `variable-val` :

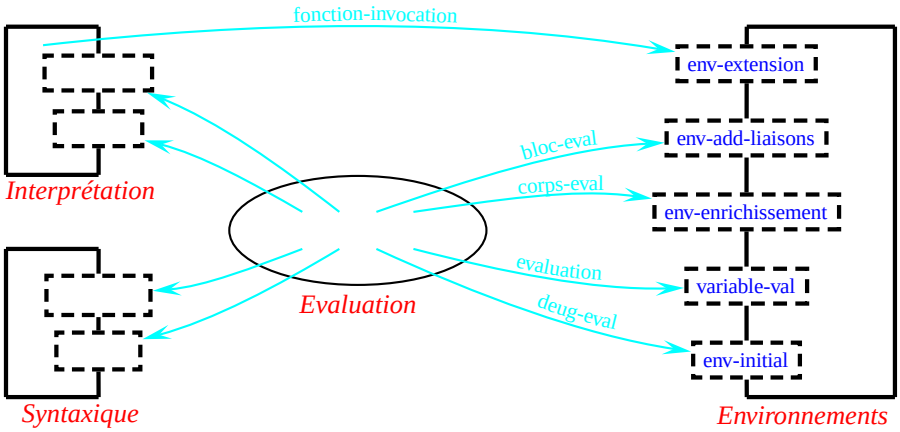


En écrivant les différents évaluateurs spécialisés, nous avons utilisé des fonctions de la barrière syntaxique et de la barrière d'interprétation ainsi que les fonctions suivantes de la barrière d'abstraction des environnements :

– la fonction `env-add-liaisons` lors de l'écriture de la fonction `bloc-eval` ,
– la fonction `env-enrichissement` lors de l'écriture de la fonction `corps-eval` :



Enfi n, lorsque nous avons implanté la barrière d'interprétation, en écrivant la défi nition de la fonction `fonction-invocation` , nous avons utilisé la fonction `env-extension` de la barrière d'abstraction des environnements :



6.2. Notion de bloc d'activation

Considérons l'« exemple » suivant :

```
(let  ((v1  exp1)
       (v2  exp2))
   (define   (f1  x1  x2  x3)
      corps-f1)
   (define   (f2  y1)
      corps-f2)
   exp-bloc)
```

a) Liaisons : ce bloc commence par des liaisons et nous devons étendre l'environnement courant en ajoutant deux associations variable – valeur.

b) Définitions fonctionnelles : nous avons ensuite deux définitions fonctionnelles et nous devons enrichir l'environnement avec deux associations variable – valeur fonctionnelle.

c) Application de fonction : enfin, dans `exp-bloc` , lors de l'application de `f1` (resp. `f2`), nous devons évaluer `corps-f1` (resp. `corps-f2`) dans l'environnement obtenu en étendant l'environnement mémorisé en liant les trois (resp. une) variables aux valeurs des trois (resp. un) arguments.

Ainsi, dans toutes ces étapes, nous devons ajouter à l'environnement courant un ensemble de couples d'associations variable – valeur : nous nommerons **bloc d'activation** un tel ensemble (noter que cette terminologie, classique en informatique, est due à l'implantation des fonctions).

6.3. Spécification de la barrière des environnements

Rappelons tout d'abord les spécifications des fonctions sur les environnements que nous avons utilisées, en commençant par deux fonctions qui vont de soi :

```
624 ;;;  env-initial: -> Environnement
    ;;;  (env-initial) rend l'environnement initial, i.e. l'environnement qui
    ;;;  contient toutes les primitives.

480 ;;;  variable-val: Variable * Environnement -> Valeur
    ;;;  (variable-val var env) rend la valeur de la variable «var» dans
    ;;;  l'environnement «env».
```

Précision

Considérons la définition suivante que nous avons déjà vue en cours (la fonction rend la liste obtenue en ajoutant l'élément donné à la fin de la liste donnée), définition où nous avions globalisé une variable :

```
(define   (&d  L x)
  (define   (&d-x  L)
    (if  (pair?  L)
         (cons  (car  L)
                (&d-x  (cdr  L)))
         (list  x)))  ; fin &d-x
  (&d-x  L))  ; fin &d
```

Dans cette définition, de quels blocs d'activation sont extraits les valeurs des différentes occurrences des variables ?

- la valeur de `L` de la dernière ligne doit être trouvée dans le bloc d'activation créé par la fonction sous-jacente liée à `&d` (la fonction externe),
- la valeur de `L` des lignes 3, 4 et 5 doit être trouvée dans le bloc d'activation, ou plus exactement dans le dernier bloc d'activation (récursivité), créé par la fonction sous-jacente liée à `&d-x` (la fonction interne),
- la valeur de `x` de l'avant dernière ligne devrait, de même, être trouvée dans le bloc d'activation créé par la fonction sous-jacente liée à `&d-x` (la fonction interne), mais, comme cette variable n'est pas présente dans ce bloc d'activation (ce n'est pas une des variables de la fonction), on doit aller la rechercher dans un bloc créé précédemment, en l'occurence dans le bloc d'activation créé par la fonction sous-jacente liée à `&d` (la fonction externe).

Continuons notre revue des fonctions de la barrière d'abstraction des environnements en étudiant les trois dernières fonctions (elles posent plus de problèmes que les deux premières), en commençant par le cas relativement simple de l'ajout d'associations variable – valeurs. Nous avons utilisé deux fonctions :

```
492 ;;; env-extension: Environnement * LISTE[Variable] * LISTE[Valeur] -> Environnement
    ;;; (env-extension env vars vals) rend l'environnement «env» étendu avec
    ;;; un bloc d'activation liant les variables «vars» aux valeurs «vals».

503 ;;; env-add-liaisons: LISTE[Liaison] * Environnement -> Environnement
    ;;; (env-add-liaisons liaisons env) rend l'environnement obtenu en ajoutant,
    ;;; à l'environnement «env», les liaisons «liaisons».
```

Rappel : nous avons eu besoin de la fonction `env-extension` pour ajouter des liaisons variable — valeur lors de l'écriture de la définition de la fonction `fonction-creation` et nous avons eu besoin de la fonction `env-add-liaisons` pour ajouter des liaisons variable — valeur lors de l'écriture de la définition de la fonction `bloc-eval`.

Remarquer que ces deux fonctions ont la même finalité mais qu'elles sont différentes de part leur signature : la fonction `env-extension` a comme données (autres que l'environnement) deux listes, la liste des variables et la liste des valeurs, alors que la fonction `env-add-liaisons` a comme donnée (autre que l'environnement) une seule liste, liste dont chaque élément est une association variable — valeur.

Enfin, nous avons eu besoin de la fonction `env-enrichissement` pour ajouter des définitions fonctionnelles lors de l'écriture de la définition de la fonction `corps-eval` – qui est appelée entre autres par `bloc-eval` :

```
516 ;;; env-enrichissement: Environnement * LISTE[Definition] -> Environnement
    ;;; (env-enrichissement env defs) rend l'environnement «env» étendu avec un
    ;;; bloc d'activation pour les définitions fonctionnelles «defs».
```

Précision : considérons l'« exemple » suivant d'un bloc où le corps possède deux définitions :

```
(let ()
  (define (f1 x)
    corps-f1)
  (define (f2 y)
    corps-f2)
  exp-bloc)
```

Pour évaluer le corps, on doit évaluer *exp-bloc* dans l'environnement obtenu en enrichissant l'environnement courant avec les deux fonctions `f1` et `f2`. Ces deux fonctions sont représentées par un 4-uplet formé du symbole `*fonction*`, de la liste des variables (i.e. `'(x)` – resp. `'(y)`), du corps de la fonction (i.e *corps-f1* – resp. *corps-f2*) et de l'environnement, `env` dans lequel on doit évaluer les applications de ces deux fonctions.

Mais quel est l'environnement `env` ? Question posée autrement : de quoi pouvons-nous nous servir dans le corps de `f1` ? Bien sûr, de l'environnement où est évalué le corps (*i.e.* celui où est évalué le bloc, plus les liaisons définies derrière le `let`), mais pas seulement : `f1` elle-même doit appartenir à l'environnement (cas de la récursivité), et aussi `f2` (et plus généralement, tous les noms des fonctions définies dans le corps).

Mais cela veut dire que lorsque nous évaluons la définition de `f1`, nous devons le faire en utilisant un environnement où il y a `f2` alors que nous ne savons même pas que `f2` existe ! Clairement, la possibilité d'utiliser quelque chose qui n'existe pas encore ne facilite pas l'écriture de l'évaluateur (de nombreux langages de programmation interdisent cela). On pourrait se dire que, pour simplifier cette écriture il suffit que, nous aussi, on l'interdise. Malheureusement, cette possibilité est absolument nécessaire si on veut pouvoir écrire des récursivités croisées (dans l'exemple si `f1` apparaît dans *corps-f2* et `f2` apparaît dans *corps-f1*). Or, par exemple, c'est exactement ce que nous avons – en plus complexe car il y en a de partout – dans le présent logiciel ; par exemple, `evaluation` appelle `alternative-eval` qui appelle `evaluation`.

6.4. Implantation (via barrière d'abstraction de bas niveau)

Afin que l'on puisse aller chercher une valeur (que ce soit de variable, que ce soit de fonction) dans un ajout antérieur, tout en donnant la priorité au derniers ajouts, on implante les environnements sous forme d'une suite de blocs d'activation, celui auquel on accède en premier étant celui qui est ajouté en dernier (en informatique, on parle de pile).

6.4.1. Définition de variable-val

L'idée de la définition de la fonction variable-val est alors très simple : on regarde si la variable est présente dans le premier bloc d'activation, auquel cas on va y chercher sa valeur et, si ce n'est pas le cas, on appelle récursivement la fonction sur le reste de l'environnement (*i.e.* l'environnement obtenu en supprimant le premier bloc d'activation). Et, bien sûr, pour que l'on puisse effectuer ce calcul, il faut qu'il y ait un bloc d'activation, autrement dit que l'environnement ne soit pas vide. D'où la définition :

```
480 ;;; variable-val: Variable * Environnement -> Valeur
    ;;; (variable-val var env) rend la valeur de la variable «var» dans
    ;;; l'environnement «env».
    (define (variable-val var env)
      (if (env-non-vide? env)
          (let ((bloc (env-1er-bloc env)))
            (let ((variables (blocActivation-variables bloc)))
              (if (member var variables)
                  (blocActivation-val bloc var)
                  (variable-val var (env-reste env)))))
          (deug-erreur 'variable-val "variable inconnue" var) ) )
```

sous réserve que l'on ait, dans la barrière d'abstraction de bas niveau, les fonctions suivantes :

```
540 ;;; env-non-vide?: Environnement -> bool
    ;;; (env-non-vide? env) rend #t ssi l'environnement «env» n'est pas vide

551 ;;; env-1er-bloc: Environnement -> BlocActivation
    ;;; ERREUR lorsque l'environnement donné est vide
    ;;; (env-1er-bloc env) rend le premier (i.e. celui qui a été ajouté en
    ;;; dernier) bloc d'activation de l'environnement «env».

558 ;;; env-reste: Environnement -> Environnement
    ;;; ERREUR lorsque l'environnement donné est vide
    ;;; (env-reste env) rend l'environnement obtenu en supprimant le premier
    ;;; bloc d'activation de l'environnement «env».
```

et, dans la barrière d'abstraction des blocs d'activation,les fonctions suivantes :

```
570 ;;; blocActivation-variables: BlocActivation -> LISTE[Variable]
    ;;; (blocActivation-variables bloc) rend la liste des variables définies
    ;;; dans le bloc d'activation «bloc»

576 ;;; blocActivation-val: BlocActivation * Variable -> Valeur
    ;;; HYPOTHESE: «var» est une variable définie dans «bloc»
    ;;; (blocActivation-val bloc var) rend la valeur de la variable «var»
    ;;; dans le bloc d'activation «bloc».
```

Pour les fonctions qui ajoutent des blocs d'activation, commençons par celle qui enrichit l'environnement avec des définitions fonctionnelles car, étant la plus complexe, c'est elle qui détermine les fonctions de base utilisées.

6.4.2. Définition de env-enrichissement

Rappels

Pour une définition de fonction, on doit lier le nom de la fonction à une fonction du Scheme sous-jacent, fonction qui est défini par :

```
464 ;;; fonction-creation: Definition * Environnement -> Fonction
    ;;; (fonction-creation definition env) rend la fonction définie par
    ;;; «definition» dans l'environnement «env».
    (define (fonction-creation definition env)
      (list '*fonction*
            (definition-variables definition)
            (definition-corps definition)
            env ) )
```

le problème était que l'environnement env de la fonction précédente doit contenir toutes les fonctions définies dans le corps pour lequel on enrichi l'environnement avec la présente définition, y compris elle-même, y compris toute fonction dont la définition suit la définition que l'on est en train d'analyser). Gros problème : pour créer les différentes fonctions on doit utiliser un environnement qui ne sera défini que lorsque toutes les fonctions seront crées ! On ne s'en sort pas ! Eh bien si (mais je pense que vous vous en doutiez) !

Notons tout d'abord que la fonction que l'on est en train de définir ne sera réelement exécutée qu'après qu'on ait défini toutes les fonctions du corps, dans la partie <expressions> de ce corps, autrement dit après que l'on ait lu toutes les informations pour définir complètement l'environnement.

Remarque : on peut dire que c'est à cause de cela qu'en Scheme les parties définitions de fonctions et expressions sont séparées dans un corps. En passant, notons que ce n'est pas le cas au toplevel qui, encore une fois, a un statut particulier et rappelons que nous n'avons pas de tel toplevel en Deug-Scheme.

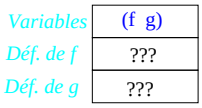
Idée

Comment allons nous faire ? Considérons le « corps » :

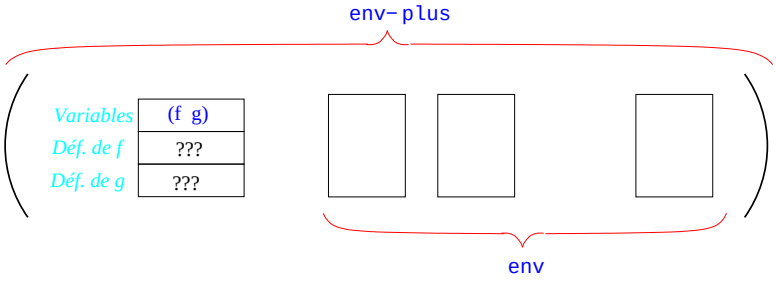
```
(define   (f ...)  ...)  ; def-f
(define   (g ...)  ...)  ; def-g
```

Dans un premier temps, nous créons un bloc d'activation qui contient :

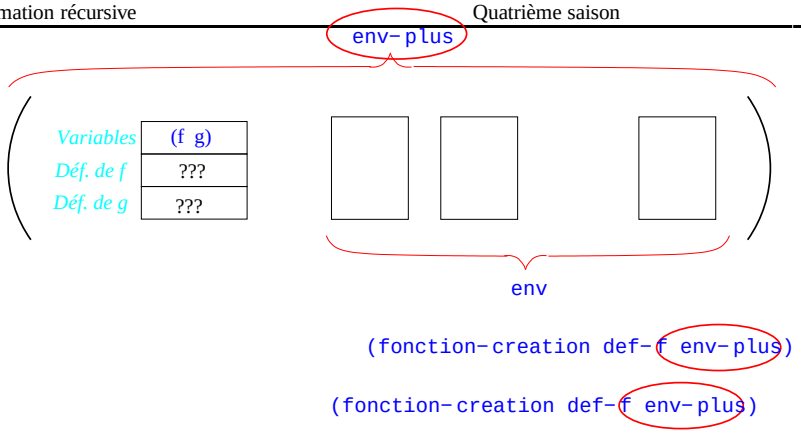
– la liste des variables de fonctions définies dans le bloc,
– autant de « cases » qu'il y a de variables définies dans le bloc, mais ces « cases » sont vides et il faudra ensuite les remplir avec la valeur des différentes fonctions (que nous calculerons en utilisant les définitions des fonctions).



Nous pouvons alors ajouter cet environnement en tête de l'environnement courant :

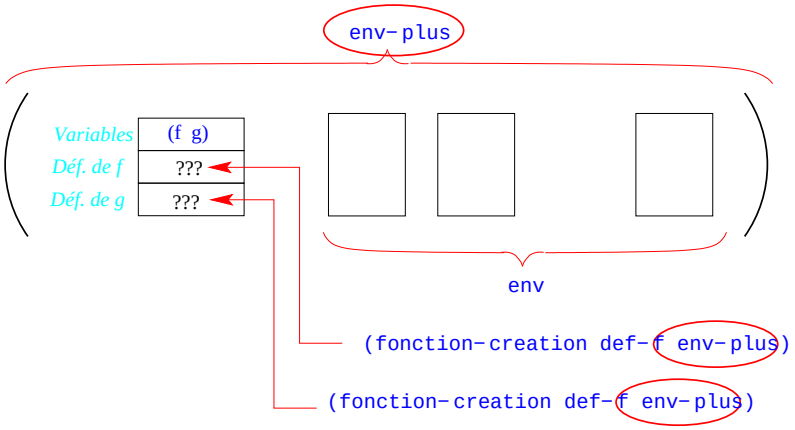


et nous pouvons alors définir les fonctions, du Scheme sous-jacent, qui implantent les deux fonctions :



Noter bien que l'environnement utilisé dans les applications de `fonction-creation` est l'environnement `env-plus` créé précédemment, environnement où il y a les différentes fonctions définies dans le corps.

Et il ne reste plus qu'à remplir les cases non remplies du bloc d'activation :



Noter que cette opération est une fonction très particulière puisqu'elle ne retourne pas de résultat (c'est comme les fonctions `display` et `newline`) et qu'elle modifie l'existant (au second semestre, on parlera de procédure). En Scheme, on parle tout de même de fonction – ou plus exactement Scheme parle toujours de procédure – mais le type du résultat est `Rien` et, par convention, le nom d'une telle fonction se termine par un point d'exclamation.

Pour mettre en œuvre cette idée, nous avons besoin des fonctions suivantes :

Fonction de la barrière d'abstraction des environnements

```
545 ;;; env-add: Environnement * BlocActivation -> Environnement
    ;;; (env-add bloc env) rend l'environnement obtenu en ajoutant devant
    ;;; l'environnement «env» le bloc d'activation «bloc».
```

Fonctions de la barrière d'abstraction des blocs d'activation

```
584 ;;; blocActivation-creation: LISTE[Variable] -> BlocActivation
    ;;; (blocActivation-creation vars) rend un bloc d'activation contenant
    ;;; la liste des variables «vars», avec la place qu'il faut pour les valeurs
    ;;; de ces variables, cette place n'étant pas remplie.

594 ;;; blocActivation-mettre-valeurs!: BlocActivation * LISTE[Valeur] -> Rien
    ;;; (blocActivation-mettre-valeurs! bloc vals) affecte les valeurs «vals» (données
    ;;; sous forme de liste) dans le bloc d'activation «bloc» (qui est un vecteur)
```

Définition de `env-enrichissement`

La définition va alors de soi, en utilisant la fonction `map` , ou plutôt `deug-map` , et une fonction interne pour pouvoir appliquer cette denière :

```
;;; env-enrichissement: Environnement * LISTE[Definition] -> Environnement
;;; (env-enrichissement env defs) rend l'environnement «env» étendu avec un
;;; bloc d'activation pour les définitions fonctionnelles «defs».
(define (env-enrichissement env defs)
  (let ((noms (deug-map definition-nom-fonction defs)))
    (let ((bloc (blocActivation-creation noms)))
      (let ((env-plus (env-add bloc env)))
        (define (fonction-creation-env-plus definition)
          (fonction-creation definition env-plus))
        (begin
          (blocActivation-mettre-valeurs!
           bloc
           (deug-map fonction-creation-env-plus defs))
          env-plus ) ) ) ) )
```

6.4.3. Définition de `env-extension`

La définition de cette fonction est plus simple que la précédente. On pourrait implanter cette fonction « d'un coup » et non en deux étapes comme ci-dessus mais cela ajouterait des fonctions de base nécessaires et comme cela se fait bien comme ça... Ainsi on fabrique l'environnement résultat en

- créant un bloc d'activation de bonne taille et où la case de la liste des variables est remplie,
- remplissant les autres cases avec les valeurs :

```
;;; env-extension: Environnement * LISTE[Variable] * LISTE[Valeur] -> Environnement
;;; (env-extension env vars vals) rend l'environnement «env» étendu avec
;;; un bloc d'activation liant les variables «vars» aux valeurs «vals».
(define (env-extension env vars vals)
  (if (= (length vars) (length vals))
      (let ((bloc (blocActivation-creation vars)))
        (begin
          (blocActivation-mettre-valeurs! bloc vals)
          (env-add bloc env) ) )
      (deug-erreur 'env-extension "arité incorrecte" (list vars vals)) ) )
```

6.4.4. Définition de `env-add-liaisons`

Nous avons déjà dit que cette fonction a la même sémantique que la précédente sauf qu'elle n'a pas la même signature (les liaisons sont données comme une liste d'associations alors que, dans la fonction prcédentes, les liaisons sont données par deux listes). Il suffit donc de fabriquer ces deux listes et d'appliquer la fonction précédente :

- la détermination de la liste des variables se fait facilement en utilisant la fonction `deug-map` appliquée sur la fonction `liaison-variable` qui, étant donnée une liaison, retourne la variable liée,
- la détermination de la liste des valeurs s'effectue en deux applications de la fonction `deug-map` :
  - la première « mappe » la liste des liaisons avec la fonction `liaison-exp` (qui, étant donnée une liaison, retourne l'expression de la valeur de la variable liée),
  - la seconde « mappe » la liste ainsi obtenue avec une fonction qui évalue ces expressions dans l'environnement donné (on a donc besoin d'une fonction interne) :

```
;;; env-add-liaisons: LISTE[Liaison] * Environnement -> Environnement
;;; (env-add-liaisons liaisons env) rend l'environnement obtenu en ajoutant,
;;; à l'environnement «env», les liaisons «liaisons».
(define (env-add-liaisons liaisons env)
  ;; eval-env : Expression -> Valeur
  ;; (eval-env exp) rend la valeur de «exp» dans l'environnement «env»
  (define (eval-env exp)
    (evaluation exp env))
  ;; expression de (env-add-liaisons liaisons env) :
  (env-extension env
                 (deug-map liaison-variable liaisons)
                 (deug-map eval-env (deug-map liaison-exp liaisons)) ) )
```

6.5. Implantation barrière d'abstraction de bas niveau

6.5.1. Rappel de la spécification

```
536 ;;; env-vide: -> Environnement
    ;;; (env-vide) rend l'environnement vide
540 ;;; env-non-vide?: Environnement -> bool
    ;;; (env-non-vide? env) rend #t ssi l'environnement «env» n'est pas vide
545 ;;; env-add: Environnement * BlocActivation -> Environnement
    ;;; (env-add bloc env) rend l'environnement obtenu en ajoutant devant
    ;;; l'environnement «env» le bloc d'activation «bloc».
551 ;;; env-1er-bloc: Environnement -> BlocActivation
    ;;; ERREUR lorsque l'environnement donné est vide
    ;;; (env-1er-bloc env) rend le premier (i.e. celui qui a été ajouté en
    ;;; dernier) bloc d'activation de l'environnement «env».
558 ;;; env-reste: Environnement -> Environnement
    ;;; ERREUR lorsque l'environnement donné est vide
    ;;; (env-reste env) rend l'environnement obtenu en supprimant le premier
    ;;; bloc d'activation de l'environnement «env».
```

6.5.2. Structure de données

Environnement = LISTE[BlocActivation]

6.5.3. Définition des fonctions

Trop facile, laissée en exercice.

6.6. Implantation barrière d'abstraction des blocs d'activation

6.6.1. Rappel de la spécification

```
570 ;;; blocActivation-variables: BlocActivation -> LISTE[Variable]
    ;;; (blocActivation-variables bloc) rend la liste des variables définies
    ;;; dans le bloc d'activation «bloc»
576 ;;; blocActivation-val: BlocActivation * Variable -> Valeur
    ;;; HYPOTHESE: «var» est une variable définie dans «bloc»
    ;;; (blocActivation-val bloc var) rend la valeur de la variable «var»
    ;;; dans le bloc d'activation «bloc».
584 ;;; blocActivation-creation: LISTE[Variable] -> BlocActivation
    ;;; (blocActivation-creation vars) rend un bloc d'activation contenant
    ;;; la liste des variables «vars», avec la place qu'il faut pour les valeurs
    ;;; de ces variables, cette place n'étant pas remplie.
594 ;;; blocActivation-mettre-valeurs!: BlocActivation * LISTE[Valeur] -> Rien
    ;;; (blocActivation-mettre-valeurs! bloc vals)  affecte les valeurs «vals» (données
    ;;; sous forme de liste) dans le bloc d'activation «bloc» (qui est un vecteur)
```

6.6.2. Structure de données

Comment créer des « cases » vides que l'on peut remplir ensuite ? Pour ce faire, il existe en Scheme la notion de vecteur. D'où :

BlocActivation = VECTEUR[LISTE[Variable] Valeur...]

Un vecteur est une suite de cases numérotées (à partir de 0) et on peut :
1 - créer un vecteur, ayant un nombre de cases donné, les cases n'étant pas remplies, avec la fonction `make-vector` :

```
;;; make-vector : nat -> VECTEUR[alpha]
;;; (make-vector taille) rend un vecteur de dimension "taille"
```

2 - affecter une valeur à une case désignée par son numéro d'ordre – rappelons que la première case a comme numéro 0 :

```
;;; vector-set! : VECTEUR[alpha] * nat * alpha -> Rien
;;; (vector-set! v i val) affecte au ("i"+1)'ème élément de "v" la
;;; valeur "val". Par exemple, "(vector-set! v 0 val)" affecte au
;;; premier élément de "v" la valeur "val".
```

Notons que cette fonction Scheme ne retourne pas de résultat et qu'elle modifie un de ses arguments. Rappelons que le type du résultat est alors *Rien* et que, par convention, le nom se termine par un point d'exclamation.

2 - connaître la valeur contenue dans une case :

```
;;; vector-ref : VECTEUR[alpha] * nat -> alpha
;;; (vector-ref v i) retourne le ("i"+1)'ème élément du vecteur "v".
;;; Par exemple, "(vector-ref v 0)" retourne le premier élément
;;; de "v", "(vector-ref v 1)" retourne le deuxième élément...
```

6.6.3. Définitions des fonctions de la barrière d'abstraction

Définition de `blocActivation-variables`

Trés simple puisque c'est le contenu de la première case du vecteur :

```
570 ;;; blocActivation-variables: BlocActivation -> LISTE[Variable]
    ;;; (blocActivation-variables bloc) rend la liste des variables définies
    ;;; dans le bloc d'activation «bloc»
    (define (blocActivation-variables bloc)
      (vector-ref bloc 0) )
```

Définition de `blocActivation-val`

L'idée est très simple : on cherche le rang, *i*, de la variable dans la liste des variables et il ne reste plus qu'à rendre le contenu de la $i^{\text{ème}}$ case du vecteur :

```
576 ;;; blocActivation-val: BlocActivation * Variable -> Valeur
    ;;; HYPOTHESE: «var» est une variable définie dans «bloc»
    ;;; (blocActivation-val bloc var) rend la valeur de la variable «var»
    ;;; dans le bloc d'activation «bloc».
    (define (blocActivation-val bloc var)
      (let ((i (rang var (blocActivation-variables bloc))))
        (vector-ref bloc i) ) )
```

Définition de `blocActivation-creation`

Très simple (ne pas oublier de remplir la case 0) :

```
584 ;;; blocActivation-creation: LISTE[Variable] -> BlocActivation
    ;;; (blocActivation-creation vars) rend un bloc d'activation contenant
    ;;; la liste des variables «vars», avec la place qu'il faut pour les valeurs
    ;;; de ces variables, cette place n'étant pas remplie.
    (define (blocActivation-creation vars)
      (let ((bloc (make-vector (+ 1 (length vars)))))
        (begin
          (vector-set! bloc 0 vars)
          bloc ) ) )
```

Définition de `blocActivation-mettre-valeurs!`

Pour assigner les valeurs, la difficulté vient du fait que l'on ne sait pas combien il y en a : on doit donc définir une fonction (interne) récursive qui remplit les cases d'un vecteur, à partir d'un indice donné, avec les éléments successifs d'une liste donnée et appliquer cette fonction avec comme indice initial 1 (premier indice où l'on trouve une valeur) et la liste de valeurs.

```
;;; blocActivation-mettre-valeurs!: BlocActivation * LISTE[Valeur] -> Rien
;;; (blocActivation-mettre-valeurs! bloc vals) affecte les valeurs «vals» (données
;;; sous forme de liste) dans le bloc d'activation «bloc» (qui est un vecteur)
(define (blocActivation-mettre-valeurs! bloc vals)
  ;; remplir!: nat * LISTE[Valeur] -> Rien
  ;; (remplir! i vals) remplit les cases du vecteur «bloc», à partir de
  ;; l'indice «i», avec les valeurs de la liste «vals» (et dans le même ordre).
  (define (remplir! i vals)
    (if (pair? vals)
        (begin
          (vector-set! bloc i (car vals))
          (remplir! (+ i 1) (cdr vals)) ) ) )
  (remplir! 1 vals) )
```

6.7. Environnement initial

Pour créer l'environnement initial il suffi t d'étendre l'environnement vide avec des liaisons nom-de-la-primitive — valeur-de-la-primitive. Pour ce faire, nous pouvons penser utiliser la fonction `env-extension` ou la fonction `env-add-liaisons`. Nous décidons d'utiliser `env-extension` (comme exercice, vous pouvez essayer de le faire avec `env-add-liaisons`; attention, une liaison est une association variable — expression et `env-add-liaisons` évalue les expressions).

C'est rrès facile (vous pouvez essayer)... sauf qu'il faut écrire (décrire) les primitives avec deux listes : la première qui contient les différents noms et la seconde qui contient les différentes valeurs, la première valeur correspondant au premier nom, la seconde valeur correspondant au second nom... la 27ème valeur correspondant au 27ème nom et gare à celui qui intervertit deux noms ou deux valeurs !

6.7.1. Description des primitives

Aussi, pour éviter des erreurs, nous décrivons chaque primitive par quatre éléments, le nom d'une primitive, la fonction correpondante du Scheme sous-jacent et l'arité représentée par un comparateur et un entier et nous fabriquerons une desciption – implantée par un n-uplet – de la primitive en utilisant la fonction `description-primitive` :

```
;;; description-primitive: Variable * (Valeur ... -> Valeur)
;;;     * (num * num -> bool) * num -> DescriptionPrimitive
;;; (description-primitive var f comparator arite) rend la description de la
;;; primitive désignée par «var», implantée dans le Scheme sous-jacent par «f» et
;;; dont l'arité est définie par «comparator» «arite».
(define (description-primitive var f comparator arite)
  (list var f comparator arite))
```

Les primitives sont alors données comme une liste de description sous une forme très lisible (voir lignes 641 à 674):

```
641 ;;; descriptions-primitives: -> LISTE[DescriptionPrimitive]
    ;;; (descriptions-primitives) rend la liste des descriptions de toutes les
    ;;; primitives
    (define (descriptions-primitives)
      (cons (description-primitive 'car car = 1)
     ...
      (cons (description-primitive 'erreur erreur >= 2)
674         '()))))))))))))))))))))))))))))))) )
```

Noter que nous avons utilisé `cons` (et non `list` qui nous aurait facilité la tâche) car, si nous avions utilisé `list` , pour l'auto-évaluation, il aurait fallu que les primitives ayant un nombre quelconque d'arguments puissent être appelées avec au moins une trentaine d'arguments (voir ce que cela implique pour l'imlantation de la fonction `primitive-invocation` ...).

6.7.2. Définition de la fonction `env-initial`

Comme nous l'avons déja dit, il faut créer les deux listes (celle des variables et celle des valeurs) à partir de cette liste. Bien sûr, nous le faisons en utilisant `deug-map` (et même deux fois pour calculer les valeurs).

```
;;; env-initial: -> Environnement
;;; (env-initial) rend l'environnement initial, i.e. l'environnement qui
;;; contient toutes les primitives.
(define (env-initial)
  (env-extension (env-vide)
                 (deug-map car (descriptions-primitives))
                 (deug-map primitive-creation
                           (deug-map cdr (descriptions-primitives)) ) ) )
```

7. Annexe : source de deug-eval

```
;;;; $Id: eval4fr.scm,v 1.12 2003/01/16 16:03:52 titou Exp $
;;;; Copyright (C) 2000 by <Titou.Durand@ufr-info-p6.jussieu.fr>
;;;;                   and <Christian.Queinnec@lip6.fr>

;;;; {{{ Grammaire du langage
;;;; Le langage interprété est défini par la grammaire suivante:
;;;;  deug-programme := expression
;;;;      expression := variable
;;;;                  | constante | (QUOTE donnée)          ; citation
;;;;                  | (COND clause *)                     ; conditionnelle
;;;;                  | (IF condition conséquence [alternant]); alternative
;;;;                  | (BEGIN expression *)                ; séquence
;;;;                  | (LET (liaison *) corps)             ; bloc
;;;;                  | (fonction argument *)               ; application
;;;;       condition := expression
;;;;     conséquence := expression
;;;;       alternant := expression
;;;;          clause := (condition expression *)
;;;;                  | (ELSE expression *)
;;;;        fonction := expression
;;;;        argument := expression
;;;;       constante := nombre | chaîne | booléen | caractère
;;;;          donnée := constante
;;;;                  | symbole
;;;;                  | (donnée *)
;;;;         liaison := (variable expression)
;;;;           corps := definition * expression expression *
;;;;      définition := (DEFINE (nom-fonction variable *) corps)
;;;;    nom-fonction := variable
;;;;        variable := tous les symboles de Scheme autres que les mots-clés
;;;;         symbole := tous les symboles de Scheme
;;;; }}} Grammaire du langage

;;;; {{{ Utilitaires généraux
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;; Nécessaires pour l'auto-amorçage (on pourrait également les placer
;;;; dans l'environnement initial).

;;; Signaler une erreur et abandonner l'évaluation.
(define (deug-erreur fn message donnee)
  (erreur 'deug-eval fn message donnee) )

;;; cadr: LISTE[alpha]/au moins deux termes/ -> alpha
;;; (cadr L) rend le second terme de la liste «L».
(define (cadr L)
  (car (cdr L)) )
```

```
;;;   cddr:  LISTE[alpha]/au     moins  deux  termes/   -> LISTE[alpha]
;;;   (cddr  L) rend  la liste  «L» privée  de ses deux premiers  termes.
(define  (cddr  L)
  (cdr  (cdr  L))  )

;;;   caddr:  LISTE[alpha]/au     moins  trois  termes/   -> alpha
;;;   (caddr  L) rend  le troisième  terme  de la liste  «L».
(define  (caddr  L)
  (car  (cdr  (cdr  L)))  )

;;;   cdddr:  LISTE[alpha]/au     moins  trois  termes/   -> LISTE[alpha]
;;;   (cdddr  L) rend  la liste  «L» privée  de ses trois  premiers  termes.
(define  (cdddr  L)
  (cdr  (cdr  (cdr  L)))  )

;;;   cadddr:  LISTE[alpha]/au     moins  quatre  termes/   -> alpha
;;;   (cadddr  L) rend  le quatrième  terme  de la liste  «L».
(define  (cadddr  L)
  (car  (cdr  (cdr  (cdr  L))))  )

;;;   length:  LISTE[alpha]    -> nat
;;;   (length  L) rend  la longueur  de la liste  «L».
(define  (length  L)
  (if  (pair?  L)
       (+ 1 (length  (cdr  L)))
       0 ) )

;;;   deug-map:  (alpha  -> beta)  * LISTE[alpha]    -> LISTE[beta]
;;;   (deug-map  f L) rend  la liste  des valeurs  de «f» appliquée  aux termes
;;;   de la liste  «L».
(define  (deug-map  f L)
  (if  (pair?  L)
       (cons  (f (car  L)) (deug-map  f (cdr  L)))
       '() ) )

;;;   member:  alpha  * LISTE[alpha]    -> LISTE[alpha]    + #f
;;;   (member  e L) rend  le suffixe  de «L» débutant  par  la première
;;;   occurrence  de «e» ou #f si «e» n'apparaît  pas dans  «L».
(define  (member  e L)
  (if  (pair?  L)
       (if  (equal?  e (car  L))
            L
            (member  e (cdr  L))  )
       #f  ) )

;;;   rang:  alpha  * LISTE[alpha]    -> nat
;;;   (rang  e L) rend  le rang  de l'élément  donné  dans  la liste  «L»
;;;   (où  on sait  que  l'élément  apparaît).  Le premier  élément  a pour  rang  un.
(define  (rang  e L)
  (if  (equal?  e (car  L))
       1
       (+ 1 (rang  e (cdr  L)))  ) )
;;;;   }}} Utilitaires  généraux

;;;;   {{{ Barrière-syntaxique
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;         ;;;;;; ;;;;;; ;;;;;; ;;;;;; ;;;;;; ;;;;;; ;;
;;;;   Ces fonctions  permettent  de manipuler  les différentes  expressions
;;;;   syntaxiques  dont Scheme  est formé.  Pour chacune  de ces différentes
;;;;   formes  syntaxiques,  on trouve  le reconnaisseur  et les accesseurs.
```

```
;;;   variable?:    Expression    -> bool
(define   (variable?    exp)
  (if  (symbol?   exp)
       (cond  ((equal?   exp  'cond)     #f)
              ((equal?   exp  'else)     #f)
              ((equal?   exp  'if)       #f)
              ((equal?   exp  'quote)    #f)
              ((equal?   exp  'begin)    #f)
              ((equal?   exp  'let)      #f)
              ((equal?   exp  'let *)    #f)
              ((equal?   exp  'define)   #f)
              ((equal?   exp  'or)       #f)
              ((equal?   exp  'and)      #f)
              (else                      #t)  )
       #f)  )

;;;   citation?:    Expression    -> bool
(define   (citation?    exp)
  (cond  ((number?    exp)   #t)
         ((string?    exp)   #t)
         ((char?   exp)      #t)
         ((boolean?    exp)  #t)
         ((pair?   exp)      (equal?   (car   exp)  'quote))
         (else               #f)  ) )

;;;   conditionnelle?:     Expression    -> bool
(define   (conditionnelle?     exp)
  (if  (pair?   exp)  (equal?   (car   exp)  'cond)   #f)  )

;;;   conditionnelle-clauses:       Conditionnelle     -> LISTE[Clause]
(define   (conditionnelle-clauses       conditionnelle)
  (cdr  conditionnelle)     )

;;;   alternative?:     Expression    -> bool
(define   (alternative?     exp)
  (if  (pair?   exp)  (equal?   (car   exp)  'if)  #f)  )

;;;   alternative-condition:       Alternative    -> Expression
(define   (alternative-condition       alt)
  (cadr   alt)  )

;;;   alternative-consequence:       Alternative    -> Expression
(define   (alternative-consequence       alt)
  (caddr   alt)  )

;;;   alternative-alternant:       Alternative    -> Expression
(define   (alternative-alternant       alt)
  (if  (pair?   (cdddr   alt))
       (cadddr   alt)
       #f  ) )

;;;   sequence?:    Expression    -> bool
(define   (sequence?    exp)
  (if  (pair?   exp)  (equal?   (car   exp)  'begin)   #f)  )

;;;   sequence-exps:     Sequence    -> LISTE[Expression]
(define   (sequence-exps     seq)
  (cdr  seq)  )

;;;   bloc?:   Expression    -> bool
```

```
(define (bloc? exp)
  (if (pair? exp) (equal? (car exp) 'let) #f) )

;;; bloc-liaisons: Bloc -> LISTE[Liaison]
(define (bloc-liaisons bloc)
  (cadr bloc) )

;;; bloc-corps: Bloc -> Corps
(define (bloc-corps bloc)
  (cddr bloc) )

;;; application?: Expression -> bool
(define (application? exp)
  (pair? exp) )

;;; application-fonction: Application -> Expression
(define (application-fonction appl)
  (car appl) )

;;; application-arguments: Application -> LISTE[Expression]
(define (application-arguments appl)
  (cdr appl) )

;;; clause-condition: Clause -> Expression
(define (clause-condition clause)
  (car clause) )

;;; clause-expressions: Clause -> LISTE[Expression]
(define (clause-expressions clause)
  (cdr clause) )

;;; liaison-variable: Liaison -> Variable
(define (liaison-variable liaison)
  (car liaison) )

;;; liaison-exp: Liaison -> Expression
(define (liaison-exp liaison)
  (cadr liaison) )

;;; definition?: Corps -> bool
;;; (definition? corps) rend #t ssi le premier élément du corps
;;; «corps» est une définition
(define (definition? corps)
  (if (pair? corps) (equal? (car corps) 'define) #f) )

;;; definition-nom-fonction: Definition -> Variable
(define (definition-nom-fonction def)
  (car (cadr def)) )

;;; definition-variables: Definition -> LISTE[Variable]
(define (definition-variables def)
  (cdr (cadr def)) )

;;; definition-corps: Definition -> Corps
(define (definition-corps def)
  (cddr def) )
;;;; }}} Barrière-syntaxique

;;;; {{{ Evaluateur
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
```

```
;;;  deug-eval:  Deug-Programme  -> Valeur
;;;  (deug-eval  p) rend la valeur du programme (de Deug-Scheme) «p».
(define (deug-eval p)
  (evaluation p (env-initial) ) )

;;;  evaluation:  Expression  * Environnement  -> Valeur
;;;  (evaluation  exp env) rend la valeur de l'expression «exp» dans
;;;  l'environnement «env».
(define (evaluation exp env)
  ;; (discrimine l'expression et invoque l'évaluateur spécialisé)
  (cond
   ((variable?  exp)        (variable-val  exp env))
   ((citation?  exp)        (citation-val  exp))
   ((alternative?  exp)     (alternative-eval
                             (alternative-condition  exp)
                             (alternative-consequence  exp)
                             (alternative-alternant  exp) env))
   ((conditionnelle?  exp)  (conditionnelle-eval
                             (conditionnelle-clauses  exp) env))
   ((sequence?  exp)        (sequence-eval  (sequence-exps  exp) env))
   ((bloc?  exp)            (bloc-eval  (bloc-liaisons  exp)
                                        (bloc-corps  exp) env))
   ((application?  exp)     (application-eval
                             (application-fonction  exp)
                             (application-arguments  exp) env))
   (else (deug-erreur  'evaluation  "pas un programme"  exp))) )

;;;  alternative-eval:  Expression  ^3 * Environnement  -> Valeur
;;;  (alternative-eval  condition consequence alternant env) rend la valeur de
;;;  l'expression «(if condition consequence alternant)» dans l'environnement «env».
(define (alternative-eval  condition consequence alternant env)
  (if (evaluation  condition env)
      (evaluation  consequence env)
      (evaluation  alternant env)) )

;;;  LISTE[Clause]  * Environnement  -> Valeur
;;;  (conditionnelle-eval  clauses env) rend la valeur, dans l'environnement «env»,
;;;  de l'expression «(cond c1 c2 ... cn)», «c1», «c2»... «cn» étant les éléments
;;;  de la liste «clauses».
(define (conditionnelle-eval  clauses env)
  (evaluation (conditionnelle-expansion  clauses) env) )

;;;  conditionnelle-expansion:  LISTE[Clause]  -> Expression
;;;  (conditionnelle-expansion  clauses) rend l'expression, écrite avec des
;;;  alternatives, équivalente à l'expression «(cond c1 c2 ... cn)»,
;;;  «c1», «c2»... «cn» étant les éléments de la liste «clauses».
(define (conditionnelle-expansion  clauses)
  (if (pair?  clauses)
      (let ((premiere-clause  (car clauses)))
        (if (equal?  (clause-condition  premiere-clause)  'else)
            (cons 'begin (clause-expressions  premiere-clause))
            (cons 'if
                  (cons (clause-condition  premiere-clause)
                        (cons (cons 'begin (clause-expressions  premiere-clause))
                              (let ((seq (conditionnelle-expansion  (cdr clauses))))
                                (if (pair?  seq)
                                    (list  seq)
                                    seq ) ) ) ) ) ) )
      '() ) )
```

```
;;;  sequence-eval:  LISTE[Expression]  * Environnement  -> Valeur
;;;  (sequence-eval  exps  env)  rend  la  valeur,  dans  l'environnement  «env»,  de
;;;  l'expression  «(begin  e1  ...  en)»,  «e1»...  «en»  étant  les éléments  de  la  liste
;;;  «exps».
;;;  (Il  faut  évaluer  tour  à  tour  les  expressions  et  rendre  la  valeur  de  la
;;;  dernière  d'entre  elles.)
(define  (sequence-eval  exps  env)
  ;;  sequence-eval+  :  LISTE[Expression]/non  vide/  -> Valeur
  ;;  même  fonction,  sachant  que  la  liste  «exps»  n'est  pas  vide  et  en globalisant
  ;;  la  variable  «env».
  (define  (sequence-eval+  exps)
    (if  (pair?  (cdr  exps))
        (begin  (evaluation  (car  exps)  env)
                (sequence-eval+  (cdr  exps))  )
        (evaluation  (car  exps)  env)))
  ;;  expression  de  (sequence-eval  exps  env)  :
  (if  (pair?  exps)
       (sequence-eval+  exps)
       #f  ) )

;;;  application-eval:  Expression  * LISTE[Expression]  * Environnement  -> Valeur
;;;  (application-eval  exp-fn  arguments  env)  rend  la  valeur  de  l'invocation  de
;;;  l'expression  «exp-fn»  aux  arguments  «arguments»  dans  l'environnement  «env».
(define  (application-eval  exp-fn  arguments  env)
  ;;  eval-env  : Expression  -> Valeur
  ;;  (eval-env  exp)  rend  la  valeur  de  «exp»  dans  l'environnement  «env»
  (define  (eval-env  exp)
    (evaluation  exp  env))
  ;;  expression  de  (application-eval  exp-fn  arguments  env)  :
  (let  ((f  (evaluation  exp-fn  env)))
    (if  (invocable?  f)
        (invocation  f  (deug-map  eval-env  arguments))
        (deug-erreur  'application-eval
                      "pas  une  fonction"  f  ) ) ) )

;;;  bloc-eval:  LISTE[Liaison]  * Corps  * Environnement  -> Valeur
;;;  (bloc-eval  liaisons  corps  env)  rend  la  valeur,  dans  l'environnement  «env»,
;;;  de  l'expression  «(let  liaisons  corps)».
(define  (bloc-eval  liaisons  corps  env)
  (corps-eval  corps  (env-add-liaisons  liaisons  env))  )

;;;  corps-eval:  Corps  * Environnement  -> Valeur
;;;  (corps-eval  corps  env)  rend  la  valeur  de  «corps»  dans  l'environnement  «env»
(define  (corps-eval  corps  env)
  (let  ((def-exp  (corps-separation-defs-exps  corps)))
    (let  ((defs  (car  def-exp))
           (exp  (cadr  def-exp)))
      (evaluation  exp  (env-enrichissement  env  defs))  ) ) )

;;;  corps-separation-defs-exps:  Corps  -> (LISTE[Definition]  * LISTE[Expression])
;;;  (corps-separation-defs-exps  corps)  rend  une  liste  dont  le  premier  élément  est
;;;  la  liste  des  definitions  du  corps  «corps»  et  les  autres  éléments  sont  les
;;;  expressions  de  ce  corps.
(define  (corps-separation-defs-exps  corps)
  (if  (definition?  (car  corps))
       (let  ((def-exp-cdr
                (corps-separation-defs-exps  (cdr  corps))))
         (cons  (cons  (car  corps)
                       (car  def-exp-cdr))
```

```
348         (cdr  def-exp-cdr)))
349         (cons  '()  corps)   ) )
350 ;;;;    }}}  Evaluateur
351
352 ;;;;    {{{  Barrière-interpretation
353 ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;          ;;;;;;  ;;;;;;  ;;;;;;  ;;;;;;  ;;;;;;  ;;;;;;  ;;
354 ;;;;  Un programme  Scheme  décrit  deux  sortes  d'objets:  les valeurs  non fonctionnelles
355 ;;;;  (les  entiers,  les booléens...  les  listes...)  et  les valeurs  fonctionnelles
356
357 ;;;;        {{{  Valeurs-non-fonctionnelles
358 ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;          ;;;;;;  ;;;;;;  ;;;;;;  ;;;;;;  ;;;;;;  ;;;;;;  ;;
359
360 ;;;   citation-val:    Citation   -> Valeur/non   fonctionelle/
361 ;;;   (citation-val    cit)  rend  la valeur  de  la citation  «cit».
362 (define   (citation-val    cit)
363   (if  (pair?  cit)
364        (cadr  cit)
365        cit  ) )
366 ;;;;        }}}  Valeurs-non-fonctionnelles
367
368 ;;;;        {{{  Valeurs-fonctionnelles
369 ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;          ;;;;;;  ;;;;;;  ;;;;;;  ;;;;;;  ;;;;;;  ;;;;;;  ;;
370 ;;;;   Il  y  a  deux  types  de  fonctions,   les  fonctions  prédéfinies  (reconnues  par
371 ;;;;   primitive?)  et  les  fonctions  du  programme  en  cours  d'évaluation  (créées  par
372 ;;;;   fonction-creation).
373
374 ;;;   invocable?:   Valeur  -> bool
375 ;;;   (invocable?   val)  rend  vrai  ssi  «val»  est  une  fonction  (primitive  ou définie
376 ;;;   par  le programmeur)
377 (define   (invocable?   val)
378   (if  (primitive?   val)
379        #t
380        (fonction?   val)  ) )
381
382 ;;;   invocation:   Invocable   * LISTE[Valeur]    -> Valeur
383 ;;;   (invocation   f vals)  rend  la valeur  de  l'application   de «f» aux éléments  de
384 ;;;   «vals».
385 (define   (invocation   f vals)
386   (if  (primitive?   f)
387        (primitive-invocation      f vals)
388        (fonction-invocation      f vals)  ) )
389
390 ;;;;            {{{  Primitives
391 ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;          ;;;;;;  ;;;;;;  ;;;;;;  ;;;;;;  ;;;;;;  ;;;;;;  ;;
392 ;;;;   Une  primitive  est  implantée  par  un  4-uplet:
393 ;;;;   - le  premier  élément  est  le  symbole  *primitive  * (pour  les  reconnaître),
394 ;;;;   - le  second  élément  est  la  fonction  du  Scheme  sous-jacent  qui  implante
395 ;;;;     la  primitive,
396 ;;;;   - le  troisième  élément  est  un  comparateur  (= ou >=),
397 ;;;;   - le  dernier  élément  est  un  entier  naturel,  ces  deux  derniers  éléments
398 ;;;;     permettant  de  spécifier  l'arité  de  la  primitive.
399
400 ;;;   primitive?:   Valeur  -> bool
401 ;;;   (primitive?   val)  rend  vrai  ssi  «val»  est  une  fonction  primitive.
402 (define   (primitive?   val)
403   (if  (pair?  val)
404        (equal?  (car  val)  '*primitive  *)
405        #f)  )
406
407 ;;;   primitive-creation:     N-UPLET[(Valeur...     -> Valeur)  (num  *  num  -> bool)  num]
```

```
408 ;;;                                  -> Primitive
409 ;;;   (primitive-creation      f-c-n)  rend  la  primitive   implantée   par  la  fonction   (du
410 ;;;   Scheme  sous-jacent)    «f»,  le  premier  élément  de  «f-c-n»,   et  dont  l'arité  est
411 ;;;   spécifiée   par  le  comparateur   «c»,  deuxième   élément   de  «f-c-n»   et  l'entier   «n»,
412 ;;;    troisième   élément   de  «f-c-n».
413 (define   (primitive-creation      f-c-n)
414   (cons  '*primitive  *  f-c-n)   )
415
416 ;;;   primitive-invocation:       Primitive   *  LISTE[Valeur]     ->  Valeur
417 ;;;   (primitive-invocation       p  vals)  rend  la  valeur  de  l'application    de  la
418 ;;;   primitive   «p»  aux  éléments   de  «vals».
419 (define   (primitive-invocation      primitive   vals)
420   (let  ((n         (length   vals))
421          (f         (cadr  primitive))
422          (compare   (caddr  primitive))
423          (arite     (cadddr  primitive)))
424     (if  (compare   n  arite)
425          (cond
426           ((=  n  0)  (f))
427           ((=  n  1)  (f  (car  vals)))
428           ((=  n  2)  (f  (car  vals)  (cadr  vals)))
429           ((=  n  3)  (f  (car  vals)  (cadr  vals)  (caddr  vals)))
430           ((=  n  4)  (f  (car  vals)  (cadr  vals)
431                           (caddr  vals)  (cadddr  vals)  ))
432           (else
433            (deug-erreur    'primitive-invocation
434                            "limite  implantation    (arités  quelconques    <  5)"
435                            vals)))
436          (deug-erreur    'primitive-invocation      "arité  incorrecte"    vals)  ) ) )
437 ;;;;             }}}  Primitives
438
439 ;;;;             {{{  Fonctions-definies      par  le  programmeur
440 ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;         ;;;;;;  ;;;;;;  ;;;;;;  ;;;;;;  ;;;;;;  ;;;;;;  ;;
441 ;;;;   Une  fonction   définie   par  le  programmeur    est  implantée   par  un  4-uplet:
442 ;;;;   -  le  premier   élément   est  le  symbole  *fonction  *  (pour   les  reconnaître),
443 ;;;;   -  le  second   élément   est  la  liste   des  variables   de  la  définition    de  la
444 ;;;;      fonction,
445 ;;;;   -  le  troisième    élément   est  le  corps   de  la  définition    de  la  fonction,
446 ;;;;   -  le  quatrième    élément   est  l'environnement     où  est  définie   la  fonction.
447
448 ;;;   fonction?:    Valeur   ->  bool
449 ;;;   (fonction?    val)  rend  vrai  ssi  «val»  est  une  fonction   créée   par  le  programmeur.
450 (define   (fonction?    val)
451   (if  (pair?   val)
452        (equal?   (car   val)   '*fonction  *)
453        #f  ) )
454
455 ;;;   fonction-invocation:       Fonction    *  LISTE[Valeur]     ->  Valeur
456 ;;;   (fonction-invocation       f  vals)  rend  la  valeur  de  l'application    de
457 ;;;   la  fonction   définie   par  le  programmeur    «f»  aux  éléments   de  «vals».
458 (define   (fonction-invocation      f  vals)
459   (let  ((variables    (cadr  f))
460          (corps        (caddr  f))
461          (env          (cadddr   f))  )
462     (corps-eval    corps   (env-extension     env  variables    vals))   ) )
463
464 ;;;   fonction-creation:      Definition    *  Environnement     ->  Fonction
465 ;;;   (fonction-creation      definition    env)  rend  la  fonction   définie   par
466 ;;;   «definition»    dans  l'environnement     «env».
467 (define   (fonction-creation      definition    env)
```

```
(list 'fonction *
          (definition-variables       definition)
          (definition-corps      definition)
          env  ) )

;;;;             }}} Fonctions-definies      par  le  programmeur
;;;;         }}} Valeurs-fonctionnelles
;;;;    }}} Barrière-interpretation

;;;;    {{{ Environnements-H      (barrière    de  haut  niveau)
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;          ;;;;;;  ;;;;;;  ;;;;;;  ;;;;;;  ;;;;;;  ;;;;;;  ;;

;;;    variable-val:     Variable    * Environnement     -> Valeur
;;;    (variable-val     var  env)  rend  la  valeur   de  la  variable   «var»   dans
;;;    l'environnement     «env».
(define   (variable-val     var  env)
   (if  (env-non-vide?     env)
        (let  ((bloc   (env-1er-bloc     env)))
          (let  ((variables     (blocActivation-variables        bloc)))
            (if  (member   var  variables)
                 (blocActivation-val      bloc  var)
                 (variable-val     var  (env-reste     env)))))
        (deug-erreur     'variable-val     "variable    inconnue"    var)  ) )

;;;    env-extension:     Environnement     * LISTE[Variable]     * LISTE[Valeur]     -> Environnement
;;;    (env-extension     env  vars  vals)  rend  l'environnement     «env»  étendu   avec
;;;    un  bloc  d'activation     liant  les  variables   «vars»   aux  valeurs   «vals».
(define   (env-extension     env  vars  vals)
   (if  (= (length   vars)  (length   vals))
        (let  ((bloc   (blocActivation-creation        vars)))
          (begin
            (blocActivation-mettre-valeurs!          bloc  vals)
            (env-add   bloc  env)  ) )
        (deug-erreur     'env-extension     "arité   incorrecte"    (list  vars  vals))   ) )

;;;    env-add-liaisons:      LISTE[Liaison]     * Environnement     -> Environnement
;;;    (env-add-liaisons      liaisons   env)  rend  l'environnement     obtenu   en  ajoutant,
;;;    à  l'environnement     «env»,   les  liaisons   «liaisons».
(define   (env-add-liaisons      liaisons   env)
   ;;   eval-env    : Expression    -> Valeur
   ;;  (eval-env    exp)  rend  la  valeur   de  «exp»  dans  l'environnement     «env»
   (define   (eval-env    exp)
     (evaluation     exp  env))
   ;;  expression    de  (env-add-liaisons      liaisons   env)  :
   (env-extension     env
                      (deug-map    liaison-variable      liaisons)
                      (deug-map    eval-env    (deug-map    liaison-exp     liaisons))    ) )

;;;    env-enrichissement:       Environnement     * LISTE[Definition]      -> Environnement
;;;    (env-enrichissement       env  defs)  rend  l'environnement     «env»  étendu   avec  un
;;;    bloc  d'activation     pour  les  définitions     fonctionnelles     «defs».
(define   (env-enrichissement       env  defs)
   (let  ((noms   (deug-map    definition-nom-fonction       defs)))
     (let  ((bloc   (blocActivation-creation        noms)))
       (let  ((env-plus    (env-add   bloc  env)))
         (define   (fonction-creation-env-plus        definition)
           (fonction-creation      definition    env-plus))
         (begin
           (blocActivation-mettre-valeurs!
            bloc
```

```
          (deug-map   fonction-creation-env-plus    defs))
          env-plus   )  )  )  )  )

;;;;        {{{  Environnements-B      (barrière   de  bas  niveau)
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;          ;;;;;;  ;;;;;;  ;;;;;;  ;;;;;;  ;;;;;;  ;;;;;;  ;;
;;;;   Les  environnements     sont  représentés    par  la  structure   de  données
;;;;   LISTE[BlocActivation]

;;;   env-vide:    ->  Environnement
;;;   (env-vide)    rend  l'environnement     vide
(define   (env-vide)    '())

;;;   env-non-vide?:     Environnement     ->  bool
;;;   (env-non-vide?     env)  rend  #t  ssi  l'environnement     «env»   n'est   pas  vide
(define   (env-non-vide?     env)
  (pair?   env)   )

;;;   env-add:   Environnement     *  BlocActivation     ->  Environnement
;;;   (env-add   bloc   env)  rend  l'environnement     obtenu   en  ajoutant   devant
;;;   l'environnement     «env»   le  bloc  d'activation    «bloc».
(define   (env-add   bloc   env)
  (cons   bloc   env)   )

;;;   env-1er-bloc:     Environnement     ->  BlocActivation
;;;   ERREUR   lorsque   l'environnement     donné   est  vide
;;;   (env-1er-bloc     env)  rend  le  premier   (i.e.  celui   qui  a  été  ajouté   en
;;;   dernier)   bloc  d'activation    de  l'environnement     «env».
(define   (env-1er-bloc     env)
    (car   env)   )

;;;   env-reste:    Environnement     ->  Environnement
;;;   ERREUR   lorsque   l'environnement     donné   est  vide
;;;   (env-reste    env)  rend  l'environnement     obtenu   en  supprimant    le  premier
;;;   bloc  d'activation    de  l'environnement     «env».
(define   (env-reste    env)
  (cdr   env)   )

;;;;             {{{  Blocs   d'activation
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;          ;;;;;;  ;;;;;;  ;;;;;;  ;;;;;;  ;;;;;;  ;;;;;;  ;;
;;;;   Les  blocs   d'activation    sont  représentés    par  la  structure   de  données:
;;;;   VECTEUR[LISTE[Variable]       Valeur...]

;;;   blocActivation-variables:        BlocActivation     ->  LISTE[Variable]
;;;   (blocActivation-variables        bloc)   rend   la  liste   des  variables    définies
;;;   dans   le  bloc   d'activation    «bloc»
(define   (blocActivation-variables        bloc)
  (vector-ref    bloc   0)  )

;;;   blocActivation-val:       BlocActivation     *  Variable   ->  Valeur
;;;   HYPOTHESE:    «var»   est  une  variable   définie   dans   «bloc»
;;;   (blocActivation-val      bloc   var)  rend   la  valeur   de  la  variable    «var»
;;;   dans   le  bloc   d'activation    «bloc».
(define   (blocActivation-val      bloc   var)
  (let  ((i  (rang   var  (blocActivation-variables        bloc))))
    (vector-ref    bloc   i)  )  )

;;;   blocActivation-creation:       LISTE[Variable]      ->  BlocActivation
;;;   (blocActivation-creation        vars)   rend   un  bloc   d'activation    contenant
;;;   la  liste   des  variables    «vars»,    avec   la  place   qu'il   faut   pour   les  valeurs
;;;   de  ces   variables,    cette   place   n'étant    pas  remplie.
```

```
588 (define   (blocActivation-creation          vars)
589   (let  ((bloc  (make-vector    (+ 1 (length  vars)))))
590     (begin
591       (vector-set!    bloc  0 vars)
592       bloc  ) ) )
593
594 ;;;   blocActivation-mettre-valeurs!:         BlocActivation    * LISTE[Valeur]    -> Rien
595 ;;;   (blocActivation-mettre-valeurs!         bloc vals)   affecte  les valeurs  «vals»  (données
596 ;;;   sous  forme  de liste)  dans  le bloc d'activation   «bloc»  (qui  est  un vecteur)
597 (define   (blocActivation-mettre-valeurs!         bloc  vals)
598   ;;   remplir!:   nat  * LISTE[Valeur]    -> Rien
599   ;;   (remplir!   i vals)  remplit  les  cases  du vecteur  «bloc»,  à  partir  de
600   ;;   l'indice  «i»,  avec  les valeurs  de la liste  «vals»  (et  dans  le même  ordre).
601   (define   (remplir!   i vals)
602     (if  (pair?  vals)
603         (begin
604           (vector-set!    bloc  i (car  vals))
605           (remplir!   (+ i 1) (cdr  vals))  ) ) )
606   (remplir!   1 vals)  )
607 ;;;;             }}} Blocs  d'activation
608 ;;;;        }}} Environnements-B     (barrière   de  bas  niveau)
609
610 ;;;;        {{{ Environnement-initial
611 ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;          ;;;;;;  ;;;;;;  ;;;;;;  ;;;;;;  ;;;;;;  ;;;;;;  ;;
612 ;;;;   L'environnement     initial  est  composé   des  primitives.
613 ;;;;   Pour  faciliter   la  description,    dans  ce  logiciel,   des  différentes
614 ;;;;   primitives,    nous  les  écrirons   sous  la  forme  d'une  liste  de
615 ;;;;   descriptions    de primitive.
616 ;;;;   Un  élément   du  type  DescriptionPrimitive      est  une  description    (presque
617 ;;;;   textuelle)   d'une  primitive.   C'est  une  liste  dont
618 ;;;;   - le  premier  élément  est  la  variable  représentant,    dans  Deug-Scheme,    la
619 ;;;;     primitive   considérée,
620 ;;;;   - les  trois  autres  éléments  sont  les  trois  éléments  qui  décrivent
621 ;;;;     l'implantation    de  la  primitive   (la  fonction   du  Scheme   sous-jacent,
622 ;;;;     le  comparateur   et  l'arité).
623
624 ;;;   env-initial:    -> Environnement
625 ;;;   (env-initial)    rend  l'environnement     initial,   i.e.  l'environnement     qui
626 ;;;   contient   toutes   les  primitives.
627 (define   (env-initial)
628   (env-extension     (env-vide)
629                      (deug-map   car  (descriptions-primitives))
630                      (deug-map   primitive-creation
631                                  (deug-map   cdr  (descriptions-primitives))       ) ) )
632
633 ;;;   description-primitive:       Variable   * (Valeur   ...  -> Valeur)
634 ;;;        * (num  * num  -> bool)  * num   -> DescriptionPrimitive
635 ;;;   (description-primitive       var  f  comparator   arite)  rend  la  description    de  la
636 ;;;   primitive   désignée   par  «var»,   implantée   dans  le  Scheme   sous-jacent    par  «f»  et
637 ;;;   dont  l'arité   est  définie   par  «comparator»    «arite».
638 (define   (description-primitive       var  f  comparator   arite)
639   (list  var  f  comparator   arite))
640
641 ;;;   descriptions-primitives:       -> LISTE[DescriptionPrimitive]
642 ;;;   (descriptions-primitives)       rend  la  liste  des  descriptions    de  toutes  les
643 ;;;   primitives
644 (define   (descriptions-primitives)
645   (list
646    (description-primitive       'car             car             =  1)
647    (description-primitive       'cdr             cdr             =  1)
```

```
    (description-primitive       'cons           cons           =  2)
    (description-primitive       'list           list           >= 0)
    (description-primitive       'vector-length  vector-length  =  1)
    (description-primitive       'vector-ref     vector-ref     =  2)
    (description-primitive       'vector-set!    vector-set!    =  3)
    (description-primitive       'make-vector    make-vector    =  1) ;  ou 2
    (description-primitive       'pair?          pair?          =  1)
    (description-primitive       'symbol?        symbol?        =  1)
    (description-primitive       'number?        number?        =  1)
    (description-primitive       'string?        string?        =  1)
    (description-primitive       'boolean?       boolean?       =  1)
    (description-primitive       'vector?        vector?        =  1)
    (description-primitive       'char?          char?          =  1)
    (description-primitive       'equal?         equal?         =  2)
    (description-primitive       '+              +              >= 0)
    (description-primitive       '*              *              >= 0)
    (description-primitive       '-              -              =  2)
    (description-primitive       '=              =              =  2)
    (description-primitive       '<              <              =  2)
    (description-primitive       '>              >              =  2)
    (description-primitive       '<=             <=             =  2)
    (description-primitive       '>=             >=             =  2)
    (description-primitive       'remainder      remainder      =  2)
    (description-primitive       'display        display        =  1) ;  ou 2
    (description-primitive       'newline        newline        =  0) ;  ou 1
    (description-primitive       'read           read           =  0)
    (description-primitive       'erreur         deug-erreur    =  3) ))

;;;;        }}} Environnement-initial
;;;;   }}} Environnements-H     (barrière   de  haut  niveau)

;;;;   {{{ Mode d'emploi
;;;; NOTA: sous DrScheme,  on doit faire tourner ce code dans un environnement  où est
;;;; définie la fonction «erreur», pour faire tourner ce code sous d'autres systèmes
;;;; Scheme, il faut définir une fonction «erreur» d'arité supérieure ou égale à 2.

;  Mise en oeuvre sous DrScheme:
;  ouvrir fichier eval4fr.scm  puis évaluer (deug-eval  '(+ 2 3))
;  puis évaluer
;  (deug-eval   '(let  ((a 5)) (define  (f n) (if (= n 0) 1 (* n (f (- n 1)))))
;                              (f a)))
;  Pour auto-interpréter     l'évaluateur,    il suffit d'écrire  (en supposant
;  que <DEUGEVAL>   est le programme  définissant   deug-eval:
;  (deug-eval   '(let  () <DEUGEVAL>   (deug-eval   '(+ 2 3))))
;  puis
;  (deug-eval   '(let  ()
;     <DEUGEVAL>
;     (deug-eval   '(let  ((a 5)) (define  (f n) (if (= n 0) 1 (* n (f (- n 1)))))
;                                 (f a)))))
;;;;   }}} Mode d'emploi
;;;;   end of eval4fr.scm
```