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

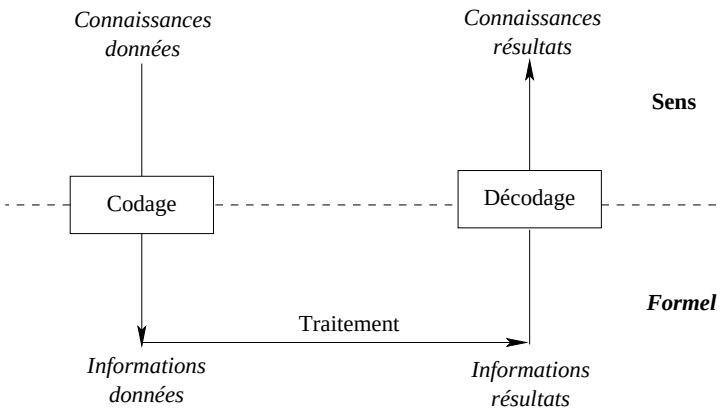
Comme pour le mot optique, le mot informatique a deux sens : c'est une science et c'est une technique. Mais alors, doit-on enseigner la science ou la technologie (d'après LE PETIT ROBERT, le sens de ce mot est « Théorie générale et études spécifiques (outils, machines, procédés...) des techniques. ») ? 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**.

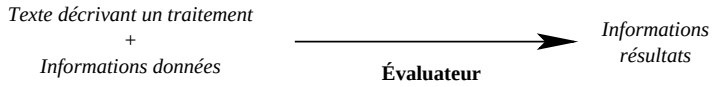
1.2. Processus d'évaluation

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.

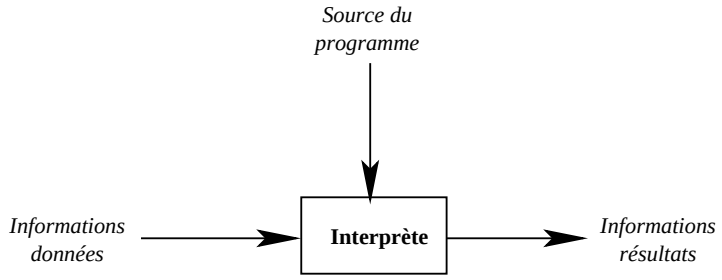
Définition : le **processus d'évaluation** permet de produire de l'information (informations résultats) à partir

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

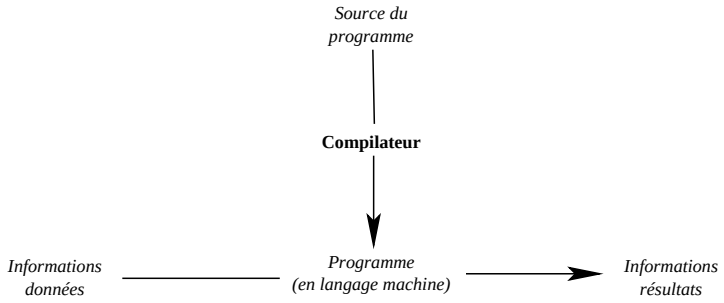


Le processus d'évaluation se décline sous deux formes :

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)
```

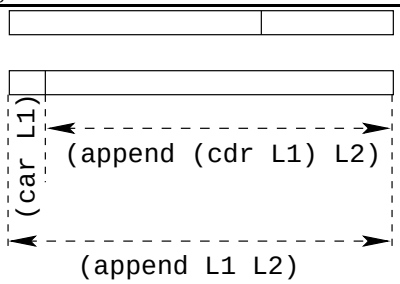
- *combinaison* $\equiv$ `cons` ,
- *base* $\equiv$ `(list x)` :

```
;;;  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))
```

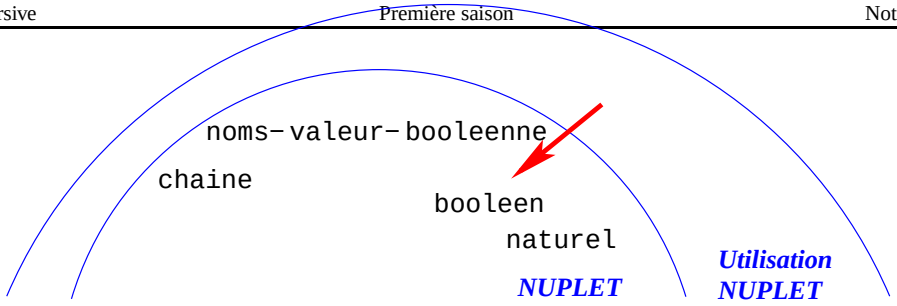
12.2. Notion de niveaux d'abstraction (premier regard)

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.

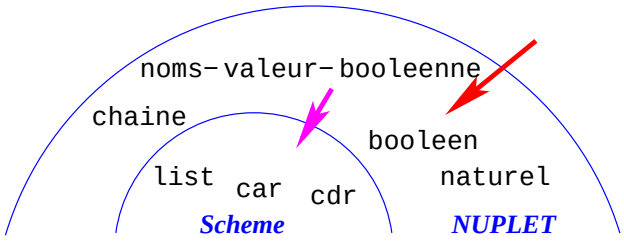
En fait, les notions de `LISTE` et de `NUPLET` n'existent pas dans Scheme. C'est nous qui les défi nissons dans le cadre de ce cours et qui les utilisons. Mais travaillant en Scheme, nous devons implanter ces notions en Scheme et, pour ce faire, nous avons utilisé la même structure de données Scheme, 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 :

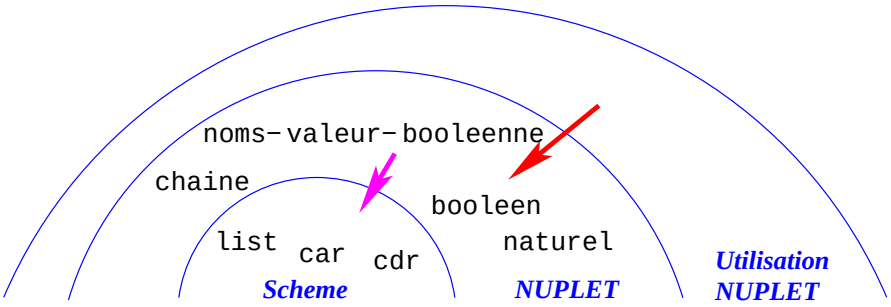


schéma qui montre bien les différents niveaux auquels on peut se placer, niveaux dans lesquels la notion de n-uplet est de plus en plus abstraite lorsque l'on s'éloigne du centre des demi-cercles.
À noter :

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*

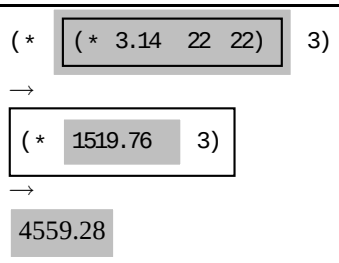
`(define (v-cylindre r h)`
`  (define (a-disque)`
`  (* 3.14 r r))`
`  (* (a-disque) h))`
	`(v-cylindre 22 3)`
	→

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))` .

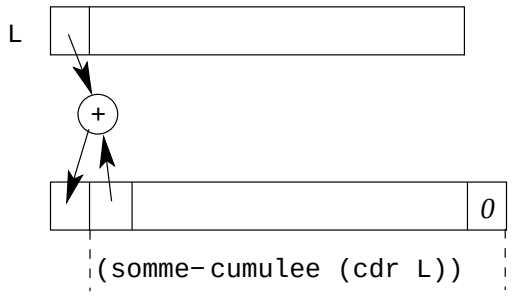
Bien sûr, dans cet exemple, le gain n'est pas très important, mais nous allons voir maintenant deux autres exemples où le fait de ne pas nommer une valeur qu'on utilise plusieurs fois peut être catastrophique pour l'efficacité.

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
"
```

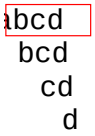
Sa spécifi cation est donc :

```
;;;  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
```

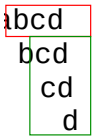
L'implantation est un peu plus délicate. Faisons un dessin : `(triangle2 "abcd")` doit rendre le paragraphe

```
bcd
bcd
 cd
  d
```

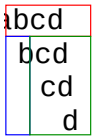
la première ligne de ce paragraphe étant la donnée auquelle on a appliqué la fonction :



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