Programmation récursive

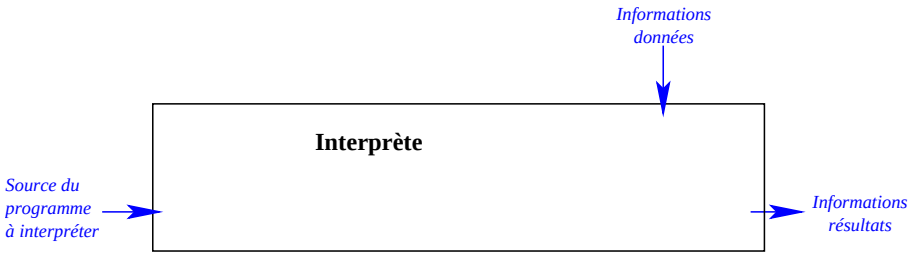
Troisième saison

Version 1.3

Sommaire

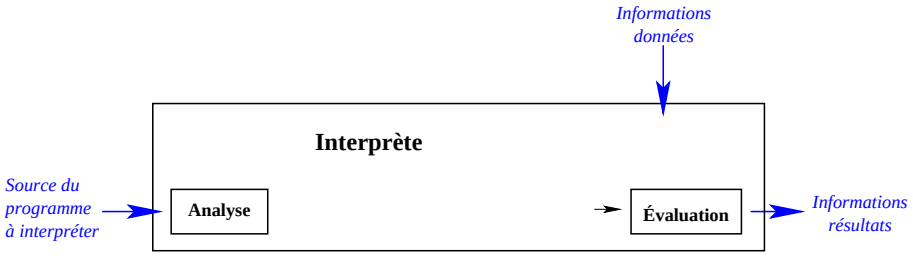
1. Introduction

Dans l'introduction de la première saison, nous avons dit qu'un interprète pouvait être schématisé par :

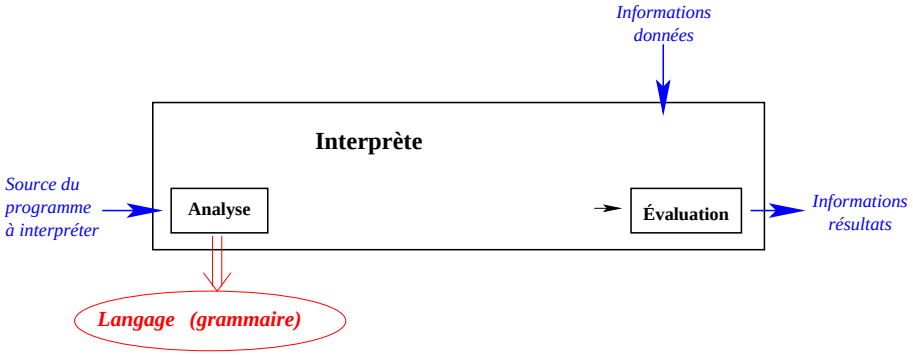


données, programmes (et résultats) étant des informations (sous forme de textes).
Remarque : dans ce cours, nous n'utilisons pratiquement pas la possibilité de lire des données (informations données).

Ainsi, un interprète doit « comprendre » le programme et les données et faire calculer par l'ordinateur le résultat en utilisant l'algorithme, ou le procédé de calcul, décrit par le programme. La « compréhension » s'effectue dans une phase dite d'*analyse*, le calcul s'effectuant dans une phase dite d'*évaluation* :

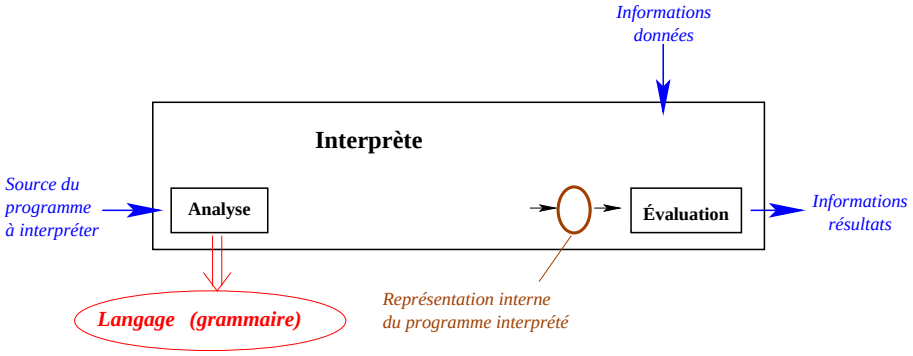


On peut analyser le source du programme parce qu'il est écrit dans un *langage* et on peut le faire analyser par un programme d'ordinateur parce que ce langage est parfaitement défi ni, en général à l'aide d'une grammaire :

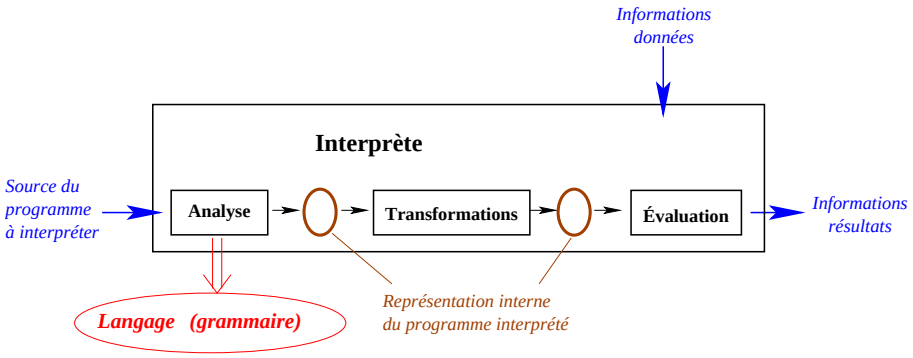


Noter que cette analyse peut être un problème diffi cile – de plus en plus lorsque l'on veut se rapprocher d'une langue naturelle – et qu'il existe des concepts et des techniques extrêmement sophistiqués pour le résoudre. Dans ce cours, nous n'analyserons que des langages simples (nous vous fournirons un analyseur pour un langage plus compliqué à analyser) mais nous aurons tout de même besoin de quelques concepts pour nous aider (nous étudierons ces concepts dans la première section).

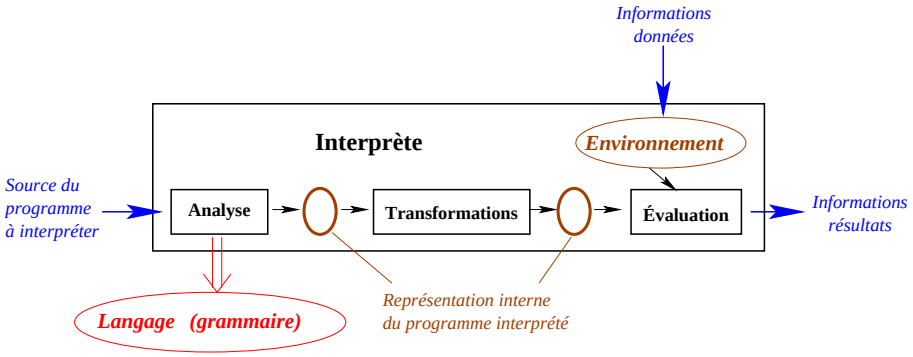
Afin que l'évaluation soit le plus efficace possible (et que le source de l'interprète correspondant soit plus simple à écrire), la phase d'analyse construit une représentation interne du programme à interpréter. Il faut bien voir que le choix de la structure de données contenant cette représentation interne est de la plus haute importance pour l'efficacité de l'interpréteur.



Pour améliorer l'efficacité de l'évaluation ou pour simplifier l'écriture de l'évaluateur (en diminuant le nombre de constructions à traiter), on peut aussi transformer la représentation du programme en une autre équivalente (c'est-à-dire dont l'évaluation donnera le même résultat) :



Enfin, dès qu'il y a la notion de variable dans le langage, l'évaluateur évalue le programme dans un *environnement*, structure de données qui, entre autres, reçoit les données du problème (noter que cet environnement évolue lors d'une évaluation) :



Dans les sections qui suivent, nous introduirons les concepts et les techniques en nous appuyant sur quatre exemples qui tournent autour des langages logiques.

2. Notion de grammaire

Avant de parler de grammaire, donnons le sens que nous utiliserons pour les mots « expression » et « langage ». Une **expression** est une représentation textuelle d'un calcul. L'ensemble des expressions utilisables est un **langage**.

2.1. Exemples de grammaires des langages des expressions booléennes simples

Comme premier exemple, étudions les expressions booléennes simples (*i.e.* la disjonction et la conjonction sont de opérateurs binaires), avec variables.

Comme nous l'avons rappelé dans le paragraphe **??**, page **??**, il existe plusieurs écritures d'une expression selon le placement de l'opérateur par rapport à ses opérandes (écriture infixe, préfixe...) et selon le status des parenthèses (écriture sans parenthèse, complètement parenthésée...). Nous donnons ci-dessous quatre définitions, à l'aide de grammaires, de langages des expressions booléennes simples.

2.1.1. En infixé, complètement parenthésée

Voici la grammaire des expressions booléennes simples avec variables en infixé complètement parenthésée :

$\langle expBoolSimple_{infixe} \rangle \rightarrow \langle atomique \rangle \mid \langle unaire \rangle \mid \langle binaire \rangle$

$\langle unaire \rangle \rightarrow$ ($\langle opérateur1 \rangle$ $\langle expBoolSimple_{infixe} \rangle$)

$\langle binaire \rangle \rightarrow$ ($\langle expBoolSimple_{infixe} \rangle$ $\langle opérateur2 \rangle$ $\langle expBoolSimple_{infixe} \rangle$)

$\langle atomique \rangle \rightarrow \langle constante \rangle \mid \langle variable \rangle$

$\langle constante \rangle \rightarrow$ `@f` FAUX | `@v` VRAI

$\langle variable \rangle \rightarrow$ *Une suite de caractères autres que l'espace, les parenthèses et @*

$\langle opérateur1 \rangle \rightarrow$ `@non`

$\langle opérateur2 \rangle \rightarrow$ `@et` ET | `@ou` OU

Terminologie :

- $\langle expBoolSimple_{infixe} \rangle$, $\langle constante \rangle$... sont des **non-terminaux** ou **unités syntaxiques** de la grammaire ; ils ne seront jamais écrits tels quels, on les dérivera (cf. plus loin) ;
- `@f`, `@v`, `@non`, `@et` et `@ou` sont des éléments **terminaux** de la grammaire ; ils sont écrits tels quels dans le langage ;
- *Une suite de caractères autres que l'espace, les parenthèses et @* est aussi un élément terminal mais, contrairement aux précédents, on le décrit de façon informelle ;
- $\langle expBoolSimple_{infixe} \rangle$, qui est la première unité syntaxique définie, est l'axiome de la grammaire : c'est d'elle dont on part pour définir un mot du langage.

Remarque : nous voulons écrire des fonctions Scheme manipulant des mots du langage, aussi avons-nous dû choisir des graphismes utilisables sur ordinateur : les opérateurs booléens classiques, $\wedge$, $\vee$ et $\neg$, ont été remplacés par `@et`, `@ou` et `@non`. D'autre part, pour bien distinguer les variables (que l'on a tendance à noter *v*) des constantes booléennes, nous avons noté ces dernières `@f` et `@v`. Noter que nous n'avons pas pris `#f` et `#t` car nous voulons bien distinguer le langage Scheme et notre langage des expressions booléennes.

2.1.2. « À la Scheme »

On peut aussi décider d'écrire les expressions « à la Scheme » (*i.e.* en préfixé, complètement parenthésées) :

$\langle expBoolSimple_{Scheme} \rangle \rightarrow \langle atomique \rangle \mid \langle unaire \rangle \mid \langle binaire \rangle$

$\langle unaire \rangle \rightarrow$ ($\langle opérateur1 \rangle$ $\langle expBoolSimple_{Scheme} \rangle$)

$\langle binaire \rangle \rightarrow$ ($\langle opérateur2 \rangle$ $\langle expBoolSimple_{Scheme} \rangle$ $\langle expBoolSimple_{Scheme} \rangle$)

$\langle atomique \rangle \rightarrow \langle constante \rangle \mid \langle variable \rangle$

<constante> → `@f` FAUX
`@v` VRAI

<variable> → *Une suite de caractères autres que l'espace, les parenthèses et @*

<opérateur1> → `@non`

<opérateur2> → `@et` ET
`@ou` OU

2.1.3. En polonaise préfixée

On peut aussi décider d'écrire les expressions en polonaise préfi xée (sans parenthèse) :

<expBoolSimple$_{polonaise}$> → *<atomique>*
<unaire>
<binaire>

<unaire> → *<opérateur1> <expBoolSimple$_{polonaise}$>*

<binaire> → *<opérateur2> <expBoolSimple$_{polonaise}$> <expBoolSimple$_{polonaise}$>*

<atomique> → *<constante>*
<variable>

<constante> → `@f` FAUX
`@v` VRAI

<variable> → *Une suite de caractères autres que l'espace, les parenthèses et @*

<opérateur1> → `@non`

<opérateur2> → `@et` ET
`@ou` OU

2.1.4. En notation « normale »

<expBoolSimple$_{normale}$> → *<atomique>*
<unaire>
<binaire>
(*<expBoolSimple$_{normale}$>*)

<unaire> → *<opérateur1> <expBoolSimple$_{normale}$>*

<binaire> → *<expBoolSimple$_{normale}$> <opérateur2> <expBoolSimple$_{normale}$>*

<atomique> → *<constante>*
<variable>

<constante> → `@f` FAUX
`@v` VRAI

<variable> → *Une suite de caractères autres que l'espace, les parenthèses et @*

<opérateur1> → `@non`

<opérateur2> → `@et` ET
`@ou` OU

Exemple 1 : `@non (@v @et @v) @ou @f`
Exemple 2 : `@non @v @et (@v @ou @f)`
Exemple 3 : `@non @v @et @v @ou @f`
Remarques :

- Pour savoir comment calculer l'expression `@non @v @et @v @ou @f`, il faut savoir qu'elle est équivalente à `((@non @v) @et @v) @ou @f`. Pour ce faire, on doit, en plus de la grammaire, donner des règles de priorités : ici, pour suivre les conventions classiques, nous décidons que le « `@non` » est plus prioritaire que le « `@et` » lui-même plus prioritaire que le « `@ou` ». Ces règles de priorités imposent de parenthéser d'abord les sous-expressions correspondant à un « `@non` » et, ensuite, celles qui correspondent à un « `@et` » (et on n'a pas besoin de parenthéser les sous-expressions qui correspondent à « `@ou` » car c'est la priorité la plus faible). Ainsi, dans l'exemple `@non @v @et @v @ou @f`,
  - il y a un « `@non` » (opérateur unaire) : cette expression est équivalente à `(@non @v) @et @v @ou @f`,

– il y a un « `@et` » (opérateur binaire) : cette expression est équivalente à `((@non @v) @et @v) @ou @f`.
– La règle

  <expBoolSimple$_{normale}$> → (*<expBoolSimple$_{normale}$>*)

  permet alors de parenthéser des sous-expressions qui, sinon, seraient découpées autrement par les règles de priorités (par exemple `@non (@v @et @v) @ou @f`.

Notes didactiques :

– ce langage est étudié parce que ce sont les expressions booléennes que vous avez l'habitude d'utiliser par ailleurs ;
– la description du langage que nous avons donné ci-dessus utilise autre chose (la notion de priorité) que la notion de grammaire ;
– il existe des grammaires auto-suffi santes pour ce langage mais elles s'éloignent de la forme des grammaires que nous avons données pour décrire les autres langages des expressions booléennes simples et, surtout, elles demandent une réflexion approfondie sur les grammaires, étude qui dépasse le cadre de cet ouvrage.

2.1.5. Similitude des différents ensembles de règles de grammaire

On peut noter que toutes ces grammaires ont des points en commun, points en commun que nousretrouverions dans tous les langages des expressions booléennes simples :

– les règles pour les expressions atomiques sont exactement les mêmes,
– surtout, les règles pour les expressions binaires disent que l'on doit décomposer l'expression avec :
  – un opérateur (qui peut être à différentes places selon le langage),
  – qui a deux opérandes qui sont des expressions (que l'on devra donc aussi décomposer),
– de même, les règles pour les expressions unaires disent toutes que l'on doit décomposer l'expression avec :
  – un opérateur (qui peut être à différentes places selon le langage),
  – qui a un opérande qui est une expression (que l'on devra donc aussi décomposer).

On peut donc parler d'une classe de langages défi nis par une classe de grammaires (ou par une classe de langages défi nie par une classe de grammaires).

2.2. Génération d'un mot du langage

Considérons un langage de la classe des langages des expressions booléennes simple, par exemple celui des expressions écrites en infi xé, complètement parenthésé.

Le langage défi ni par une grammaire est égal à l'ensemble des mots (ou, si vous préférez, expressions) que l'on peut générer à l'aide de la grammaire.

Notation : dans les schémas de ce paragraphe, pour des raisons de mise en page, les unités syntaxiques *<expBoolSimple$_{infixe}$>*, *<atomique>*, *<constante>*, *<opérateur1>* et *<opérateur2>* sont renommées respectivement *<expr>*, *<atom>*, *<cste>*, *<op1>* et *<op2>*.

Pour générer un mot du langage à partir de la grammaire, on part de l'axiome (rappelons que c'est la première unité syntaxique défi nie dans la grammaire), ici *<expBoolSimple$_{infixe}$>* que nous avons renommé *<expr>* :

expr

et on la dérive, c'est-à-dire que l'on prend une des possibilités de la règle et qu'on remplace l'axiome par le membre droit de cette possibilité. Par exemple, en prenant la troisième possibilité :

binaire

<binaire> est un non-terminal : on doit effectuer la dérivation (il n'y a qu'une possibilité) :

(*expr* *op2* *expr*)

les deux parenthèses sont des terminaux : on ne les touche pas ; en revanche, on dérive comme ci-dessus les non-terminaux *<expr>*, *<op2>* et *<expr>* (remarquez que l'axiome joue le même rôle que n'importe quel non-terminal, sauf au départ). Prenons la deuxième possibilité pour le premier *<expr>*, la seconde possibilité pour *<op2>* et la première possibilité pour le second *<expr>* :

unaire *@ou* *atom*

@et est un terminal : on ne le touche pas ; dérivons les terminaux *<unaire>* (il n'y a qu'une posibilité) et *<atom>* (qui correspond à *<atomique>*) en prenant la première possibilité :

pour <*op1*>, il n'y a qu'une possibilité ; pour <*expr*>, décidons de prendre la troisième possibilité ; pour <*cste*> (qui correspond à <*constante*>), prenons la première possibilité :

$\overbrace{@non}\quad \overbrace{binaire}\qquad\qquad \overbrace{@f}$

le seul non-terminal est <*binaire*> et il n'y a qu'une possibilité :

$\overbrace{(\quad expr \quad op2 \quad expr \quad)}$

décidons de prendre la première possibilité pour les deux occurrences de <*expr*> et la première possibilité pour <*op2*> :

$\overbrace{atom}\quad \overbrace{@et}\quad \overbrace{atom}$

décidons de prendre la première possibilité pour les deux occurrences de <*atom*> :

$\overbrace{cste}\qquad \overbrace{cste}$

et enfi n dérivons les deux occurrences du non-terminal <*cste*> en prenant la seconde possibilité dans les deux cas :

$\overbrace{@v}\qquad \overbrace{@v}$

Toutes les dérivations ont été effectuées (il ne reste plus de non-terminaux en suspens). Pour connaître le mot généré, il suffi t de lire de gauche à droite les terminaux :

$($ $($ *@non* $($ *@v* *@et* *@v* $)$ $)$ *@ou* *@f* $)$

2.2.1. Constructeurs

Par programme, on ne génère pas l'ensemble des mots du langage, mais on construit souvent des mots du langage (par exemple, dans la suite de cet ouvrage, on simplifi era des expressions : il s'agit de construire une expression à partir d'une expression donnée).

Pour ce faire, on doit pouvoir effectuer exactement ce que nous avons fait « à la main » lorsque nous avons généré un mot à partir de la grammaire. Autrement dit, nous devons pouvoir écrire des constantes (vrai et faux), des variables, des opérateurs (non, et, ou) et des expressions composées (unaires et binaires), ce qui sera fait en utilisant des **constructeurs** (`ebs-constante-vrai`, `ebs-constante-faux`, `ebs-variable`, `ebs-unaire`, `ebs-binaire`, `ebs-operateur1-non`, `ebs-operateur2-et` et `ebs-operateur2-ou`).

Règles de nomenclature : pour faciliter la lecture des programmes (et donc leur écriture), il est important de se donner, et de suivre, des règles pour le choix des noms des fonctions. Nous préciserons plus loin les différentes règles que nous suivons, mais notons tout de suite que, systématiquement, les noms des fonctions utiles pour un langage seront préfi xés par le nom abrégé du langage (ici « ebs- » pour « expressions booléennes simples »).

2.3. Analyse d'un mot

Les grammaires sont utilisées, particulièrement en informatique, pour savoir si un mot (une phrase) appartient bien à un certain langage (penser aux langages de programmation, aux langages de commande...) et pour effectuer une certaine tâche lorsque c'est le cas. On nomme **analyse d'un mot** la décomposition de ce mot, décomposition effectuée pour savoir si le mot appartient au langage ou pour déterminer la tâche qu'il exprime.

Dans ce paragraphe, nous voudrions voir comment écrire un programme qui réalise cette analyse. Pour ce faire, regardons comment on pourrait analyser « à la main » un mot du langage, en dégageant les fonctions qui seront nécessaires pour écrire un tel programme d'analyse.

2.3.1. Reconnaisseurs

Supposons que l'on veuille savoir si

$$((@non\ (@v\ @et\ @v))\ @ou\ @f)$$

est un mot du langage engendré par la grammaire précédente. Pour que ce soit le cas, il faut qu'il dérive de l'axiome <*expBoolSimple*$_{infixe}$> :

$$??\ <expBoolSimple_{infixe}> \Rightarrow ((@non\ (@v\ @et\ @v))\ @ou\ @f)\ ??$$

La règle de <*expBoolSimple*$_{infixe}$> fournit trois possibilités. Laquelle choisir ? Ici,

on voit facilement que le mot ne peut pas dériver du non-terminal *<atomique>* (car les mots qui dérivent de ce non terminal ne commencent pas par une parenthèse ouvrante),
- on peut voir qu'il ne peut pas dériver du non-terminal *<unaire>* (car le second caractère d'un tel mot est toujours un « @ »),
- on peut voir qu'il est possible qu'il dérive du non-terminal *<binaire>*.

Si l'on veut écrire un programme qui analyse un mot afin de déterminer s'il appartient au langage engendré par la grammaire, il faut avoir des fonctions pour décider dans quel cas on est. On nomme ces fonctions – qui sont des prédicats – des **reconnaisseurs**.

Par exemple, avec notre grammaire des expressions booléennes simples, nous devons avoir les reconnaisseurs `ebs-atomique?`, `ebs-unaire?` et `ebs-binaire?`.

Attention, en général, on ne peut pas demander que les reconnaisseurs soient infaillibles, tout en exigeant que le programme d'analyse, lui, le soit. Pour s'en convaincre il suffit de se dire que la définition de la fonction d'analyse serait alors triviale (or cela se saurait si la définition des fonctions d'analyse était triviale : c'est un problème qui a occupé de nombreux chercheurs dans les années 50 et 60) :

```
(define (ebs-infixe? m)
  (or (ebs-atomique? m) (ebs-unaire? m) (ebs-binaire? m)))
```

Ce qu'on demande aux reconnaisseurs, c'est qu'ils nous aiguillent vers la bonne règle lorsque le mot appartient au langage (il serait peut-être plus judicieux de les appeler « aiguilleurs » mais classiquement on les nomme reconnaisseurs). Ainsi, en général, les corps des définitions de fonctions dont la donnée est `exp`, supposé du langage, seront de la forme :

```
(cond ((ebs-atomique? exp)
       ; expression de la valeur à rendre lorsque exp dérive de <atomique>
        )
      ((ebs-unaire? exp)
       ; expression de la valeur à rendre lorsque exp dérive de <unaire>
        )
      ((ebs-binaire? exp)
       ; expression de la valeur à rendre lorsque exp dérive de <binaire>
        )
      (else (erreur 'fn "donnée pas bien formée")))
```

2.3.2. Accesseurs

Revenons à l'analyse, « à la main », du mot $((@non\ (@v\ @et\ @v))\ @ou\ @f)$. Nous avons dit que ce mot ne pouvait dériver que du non-terminal *<binaire>*. Regardons si c'est possible :

$??\ <binaire> \Rightarrow ((@non\ (@v\ @et\ @v))\ @ou\ @f)\ ??$

La règle de *<binaire>* ne comporte qu'une possibilité : il n'y a donc pas de problème de choix. Le mot et la partie droite commencent par une parenthèse ouvrante (cela correspond donc) et elles finissent par une parenthèse fermante (cela correspond toujours) mais, dans la partie droite de la règle, il reste $<expBoolSimple_{infixe}>$ *<opérateur2>* $<expBoolSimple_{infixe}>$ qu'il faut faire correspondre avec $(@non\ (@v\ @et\ @v))\ @ou\ @f$.

À la main, on voit que la décomposition doit être $(@non\ (@v\ @et\ @v))$ pour la première occurrence de $<expBoolSimple_{infixe}>$, *@ou* pour *<opérateur2>* et $@f$ pour la seconde occurrence de $<expBoolSimple_{infixe}>$.

Si l'on veut écrire un programme il faut que l'on ait des fonctions pour accéder aux différents éléments de cette décomposition. On nomme **accesseurs** de telles fonctions. Ainsi, dans notre exemple, nous avons besoin des accesseurs `ebs-binaire-operande-gauche`, `ebs-binaire-operande-droit` et `ebs-binaire-operateur`.

Et on continue en analysant $(@non\ (@v\ @et\ @v))$ (qui doit dériver de $<expBoolSimple_{infixe}>$), *@ou* (qui doit dériver de *<opérateur2>*) et $@f$ (qui doit dériver de $<expBoolSimple_{infixe}>$) :

$??\ <expBoolSimple_{infixe}> \Rightarrow (@non\ (@v\ @et\ @v))\ ??$
...
$??\ <opérateur2> \Rightarrow @ou\ ??$
...
$??\ <expBoolSimple_{infixe}> \Rightarrow @f\ ??$
...

3. Notion de barrière syntaxique

Nous avons vu que les différentes grammaires des expressions booléennes simples définissaient une classe de langages. Aussi les spécifications des reconnaisseurs, des accesseurs et des constructeurs seront exactement les mêmes dans tous les cas (si vous ne nous croyez pas – ce qui serait plus que légitime –, redéfinissez-les pour les expressions préfixes).

D'autre part, nous avons vu que pour analyser un mot d'un langage des expressions booléennes simples, il fallait avoir

- les reconnaisseurs `ebs-atomique?`, `ebs-variable?`, `ebs-constante?`, `ebs-constante-vrai?`, `ebs-constante-faux?`, `ebs-unaire?`, `ebs-binaire?`, `ebs-operateur1?`, `ebs-non?`, `ebs-operateur2?`, `ebs-operateur2-et?`, `ebs-operateur2-ou?`,
- les accesseurs
  - `ebs-unaire-operande`, `ebs-unaire-operateur`,
  - `ebs-binaire-operande-gauche`, `ebs-binaire-operande-droit` et `ebs-binaire-operateur`.

et que pour construire des expressions booléennes simples, il fallait avoir

- les constructeurs `ebs-variable`, `ebs-constante-vrai`, `ebs-constante-faux`, `ebs-unaire`, `ebs-binaire`, `ebs-operateur1-non`, `ebs-operateur2-et` et `ebs-operateur2-ou`.

Il est clair que l'implantation de ces différentes fonctions est interdépendante et dépend du choix que l'on fait pour implanter les expressions. On doit donc regrouper cet ensemble de fonctions dans une barrière d'abstraction, barrière d'abstraction pour une classe de grammaires, qu'on appelle la **barrière syntaxique**.

Nous allons donner la spécification précise de ces fonctions après quelques généralités.

3.1. Schéma des spécifications des fonctions ayant comme donnée un mot du langage

Systématiquement, nous nommerons le type des mots qui dérivent d'un non-terminal par le nom de celui-ci (sans accents), avec la première lettre en majuscule. Par exemple, le type des expressions représentés par des mots dérivant de $<expBoolSimple_{infixe}>$ sera noté `ExpBoolSimple`, le type des mots dérivant de *<unaire>* sera noté `Unaire`..., et la spécification des fonctions qui ont comme données un tel type est du genre :

```
;;; fn1: ExpBoolSimple -> ...
;;; (fn1 exp) rend ...

;;; fn2: Unaire -> ...
;;; (fn2 exp) rend ...
```

Rappelons qu'une telle spécification signifie que l'on garantit que la fonction rend bien ce qui est dit après « rend » lorsque la donnée appartient au type donné (`ExpBoolSimple` dans le cas de `fn1`, `Unaire` dans le cas de `fn2`) et que l'on ne sait pas ce qu'elle fait (signifie une erreur ou rend une valeur plus ou moins significative) dans le cas contraire. On pourra donner des précisions dans la spécification :

- si nous savons qu'une erreur est signifiée sous certaines conditions, nous l'indiquerons dans la spécification comme d'habitude :

  ```
  ;;; fn3: ExpBoolSimple -> ...
  ;;; ERREUR lorsque ...
  ;;; (fn3 exp) rend ...
  ```

- pour de nombreuses fonctions, nous saurons quelle valeur elles rendent lorsque la donnée n'appartient pas au type ; nous l'indiquerons alors explicitement dans la spécification :

  ```
  ;;; fn4: ExpBoolSimple -> ...
  ;;; (fn4 exp) rend ...
  ;;; (elle rend ... lorsque ...)
  ```

(nous verrons des exemples concrets ci-après).

Remarque : ces précisions sont particulièrement utiles pour les reconnaisseurs car elles permettent de donner des messages d'erreur plus significatifs lorsque le mot à analyser n'appartient pas au langage.

3.2. Spécifications des reconnaisseurs

Noter que lorsqu'on applique les reconnaisseurs `ebs-atomique?`, `ebs-unaire?` et `ebs-binaire?`, on espère que le mot à analyser dérive du non-terminal *<expBoolSimple$_{infixe}$>*. Le type de la donnée de ces reconnaisseurs est donc `ExpBoolSimple` (et le type d'arrivée est bien sûr `bool`).

Pour `ebs-atomique?`, nous avons dit qu'il était aisé de savoir si un mot quelconque pouvait dériver ou non de *<atomique>*. On peut donc prendre comme spécifi cation :

```
;;; ebs-atomique? : ExprBoolSimple -> bool
;;; (ebs-atomique? exp) rend #t ssi exp est une expression atomique
;;; (constante ou variable)
```

Pour `ebs-unaire?`, nous avons dit qu'il était diffi cile, voire impossible, de savoir si un mot quelconque pouvait dériver ou non de *<unaire>*. En revanche, il peut être aisé de savoir si on peut bien le décomposer en deux composants. D'où la spécifi cation :

```
;;; ebs-unaire? : ExprBoolSimple -> bool
;;; (ebs-unaire? exp) rend #t ssi exp ne peut être qu'une expression dérivant
;;; de <unaire>
;;; (elle rend #f lorsque exp n'est pas composée de deux composants)
```

De même, pour `ebs-binaire?`, il est aisé de savoir si on peut bien le décomposer en trois composants. D'où la spécifi cation :

```
;;; ebs-binaire? : ExprBoolSimple -> bool
;;; (ebs-binaire? exp) rend #t ssi exp ne peut être qu'une expression
;;; dérivant de <binaire>
;;; (elle rend #f lorsque exp n'est pas composée de trois composants)
```

Lorsqu'on applique les reconnaisseurs `ebs-variable?` et `ebs-constante?`, on espère que le mot à analyser dérive du non-terminal *<atomique>*. Le type de la donnée de ces reconnaisseurs est donc `Atomique` :

```
;;; ebs-variable? : Atomique -> bool
;;; (ebs-variable? exp) rend #t ssi exp est une variable
```

```
;;; ebs-constante? : Atomique -> bool
;;; (ebs-constante? exp) rend #t ssi exp est une constante
;;; (rend #f lorsque exp n'est pas une expression atomique)
```

Noter que la spécifi cation de la fonction `ebs-constante?` que nous avons donnée indique qu'en fait ce reconnaisseur rend toujours (y compris lorsque l'on ne sait pas que le mot doit dériver de *<atomique>*) la bonne réponse.

Donnons les spécifi cations des autres reconnaisseurs sans autres commentaires :

```
;;; ebs-constante-vrai? : Constante -> bool
;;; (ebs-constante-vrai? exp) rend #t ssi exp est la constante @V
;;;   (rend #f lorsque exp n'est pas une constante)
```

```
;;; ebs-constante-faux? : Constante -> bool
;;; (ebs-constante-faux? exp) rend #t ssi exp est la constante @F
;;;   (rend #f lorsque exp n'est pas une constante)
```

```
;;; ebs-operateur1? : Symbole -> bool
;;; (ebs-operateur1? s) rend #t ssi s est un opérateur unaire
```

```
;;; ebs-operateur2? : Symbole -> bool
```

```
;;; (ebs-operateur2? s) rend #t ssi s est un opérateur binaire
```

```
;;; ebs-operateur1-non? : Symbole -> bool
;;; (ebs-operateur1-non? s) rend #t ssi s est l'opérteur (unaire) non
```

```
;;; ebs-operateur2-et? : Symbole -> bool
;;; (ebs-operateur2-et? s) rend #t ssi s est l'opérteur (binaire) et
```

```
;;; ebs-operateur2-ou? : Symbole -> bool
;;; (ebs-operateur2-ou? s) rend #t ssi s est l'opérteur (binaire) ou
```

3.3. Spécifications des accesseurs

Noter que lorqu'on applique les accesseurs `ebs-unaire-operande` et `ebs-unaire-operateur`, on espère que le mot à analyser dérive du non-terminal *<unaire>*. Le type de la donnée de ces reconnaisseurs est donc `Unaire`. Le type d'arrivée est le type qui correspond au non-terminal extrait. Par exemple, la règle étant

<unaire> → (*<opérateur1>* $<expBoolSimple_{infixe}>$)

`ebs-unaire-operateur` doit rendre un élément de `Operateur1` et `ebs-unaire-operande` doit rendre un élément de `expression`. Noter aussi que l'on ne peut rien dire lorsque le mot donné ne dérive pas de *<unaire>*. Les spécifications sont donc :

```
;;; ebs-unaire-operateur : Unaire -> Operateur1
;;; (ebs-unaire-operateur exp) rend l'opérateur (principal) de l'expression
;;; donnée
```

```
;;; ebs-unaire-operande : Unaire -> ExprBoolSimple
;;; (ebs-unaire-operande) rend l'expression, opérande de l'expression donnée
```

De même, les spécifications des accesseurs correspondant à la règle

<binaire> → ($<expBoolSimple_{infixe}>$ *<opérateur2>* $<expBoolSimple_{infixe}>$)

sont :

```
;;; ebs-binaire-operande-gauche : Binaire -> ExprBoolSimple
;;; (ebs-binaire-operande-gauche exp) rend l'expression, opérande gauche de
;;; l'expression donnée
```

```
;;; ebs-binaire-operande-droit : Binaire -> ExprBoolSimple
;;; (ebs-binaire-operande-droit exp) rend l'expression, opérande droit de
;;; l'expression donnée
```

```
;;; ebs-binaire-operateur : Binaire -> Operateur2
;;; (ebs-binaire-operateur exp) rend l'opérateur (principal) de l'expression
;;; donnée
```

Remarque : noter la règle de nommage des fonctions que nous utilisons (un préfixe correspondant au non-terminal analysé, un suffixe correspondant à l'unité syntaxique extraite).

3.4. Spécifications des constructeurs

```
;;;; Constructeurs:

;;; ebs-variable : Symbole -> ExprBoolSimple
;;; (ebs-variable s) rend l'expression booléenne simple réduite à la variable s


;;; ebs-constante-vrai : -> Constante
;;; (ebs-constante-vrai) rend la constante « vrai »


;;; ebs-constante-faux : -> Constante
;;; (ebs-constante-faux) rend la constante « faux »


;;; ebs-unaire : Operateur1 * ExprBoolSimple -> ExprBoolSimple
;;; (ebs-unaire op exp) rend l'expression composée unaire ayant comme
;;; opérateur op et comme opérande exp


;;; ebs-binaire : ExprBoolSimple * Operateur2 * ExprBoolSimple -> ExprBoolSimple
;;; (ebs-binaire exp1 op exp2) rend l'expression composée ayant comme
;;; opérande gauche exp1, comme opérateur op et comme opérande droit
;;; exp2


;;; ebs-operateur1-non : -> Operateur1
;;; (ebs-operateur1-non) rend l'opérateur de négation


;;; ebs-operateur2-et : -> Operateur2
;;; (ebs-operateur2-et) rend l'opérateur de conjonction


;;; ebs-operateur2-ou : -> Operateur2
;;; (ebs-operateur2-ou) rend l'opérateur de disjonction
```

Exemple : pour construire l'expression booléenne, représentée en notation infi xée, complètement parenthésée, par :

((@non (@v @et @v)) @et (@v @ou @f))

on peut écrire :

```
(ebs-binaire (ebs-unaire (ebs-operateur1-non)
                         (ebs-binaire (ebs-constante-vrai)
                                      (ebs-operateur2-et)
                                      (ebs-constante-vrai)))
             (ebs-operateur2-et)
             (ebs-binaire (ebs-constante-vrai)
                          (ebs-operateur2-ou)
                          (ebs-constante-faux)))
```

3.5. Premier exemple d'utilisation de la barrière syntaxique

Écrivons la défi nition d'une fonction qui teste si une expression booléenne simple est une expression constante (*i.e.* qui ne comporte pas de variable).

Spécification

La spécifi cation de cette fonction est :

```
;;; ebs-exp-cste? : ExprBoolSimple -> bool
```

;;; *(ebs-exp-cste? exp) rend #t ssi exp est une expression constante*
;;; *(c'est-à-dire qui n'a pas d'occurrence de variable)*

Exemple d'application

```
(let ((expression
       (ebs-binaire (ebs-unaire (ebs-operateur1-non)
                                (ebs-constante-faux))
                    (ebs-operateur2-et)
                    (ebs-binaire (ebs-variable 'a)
                                 (ebs-operateur2-ou)
                                 (ebs-variable 'b)))))
  (ebs-exp-cste? expression))→#f
```

Implantation

Comme pour toutes les fonctions dont la donnée est un mot du langage, la forme générale de la définition de `ebs-exp-cste?` est une conditionnelle qui passe en revue toutes les branches de la règle donnée pour l'axiome (rappelons que c'est la première règle) :

```
(define (ebs-exp-cste? exp)
  (cond
    ((ebs-atomique? exp)
     ... à faire
    ((ebs-unaire? exp)
     ... à faire
    ((ebs-binaire? exp)
     ... à faire
     ... à faire
    (else (erreur 'ebs-exp-cste?
                  "expression mal formée"))))
```

et il n'y a plus qu'à remplir les trous. Lorsque l'expression est une expression atomique, elle est constante si, et seulement si, le reconnaisseur `ebs-constante?` rend `#t` :

```
(define (ebs-exp-cste? exp)
  (cond
    ((ebs-atomique? exp)
     (ebs-constante? exp))
    ((ebs-unaire? exp)
     ... à faire
    ((ebs-binaire? exp)
     ... à faire
     ... à faire
    (else (erreur 'ebs-exp-cste?
                  "expression mal formée"))))
```

lorsque l'expression donnée est une expression unaire, elle est constante si, et seulement si, son opérande est une expression constante :

```
(define (ebs-exp-cste? exp)
  (cond
    ((ebs-atomique? exp)
     (ebs-constante? exp))
    ((ebs-unaire? exp)
     (ebs-exp-cste? (ebs-unaire-operande exp)))
    ((ebs-binaire? exp)


    (else (erreur 'ebs-exp-cste?
                  "expression mal formée"))))
```

lorsque l'expression donnée est une expression binaire, elle est constante si, et seulement si, ses deux opérandes sont

des expressions constantes :

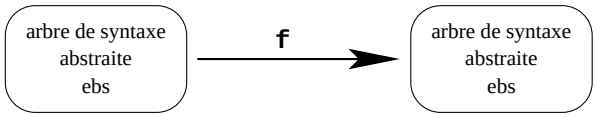
```
(define (ebs-exp-cste? exp)
  (cond
    ((ebs-atomique? exp)
     (ebs-constante? exp))
    ((ebs-unaire? exp)
     (ebs-exp-cste? (ebs-unaire-operande exp)))
    ((ebs-binaire? exp)
     (and (ebs-exp-cste? (ebs-binaire-operande-gauche exp))
          (ebs-exp-cste? (ebs-binaire-operande-droit exp))))
    (else (erreur 'ebs-exp-cste?
                  "expression mal formée"))))
```

4. Fonctions de lecture et d'écriture

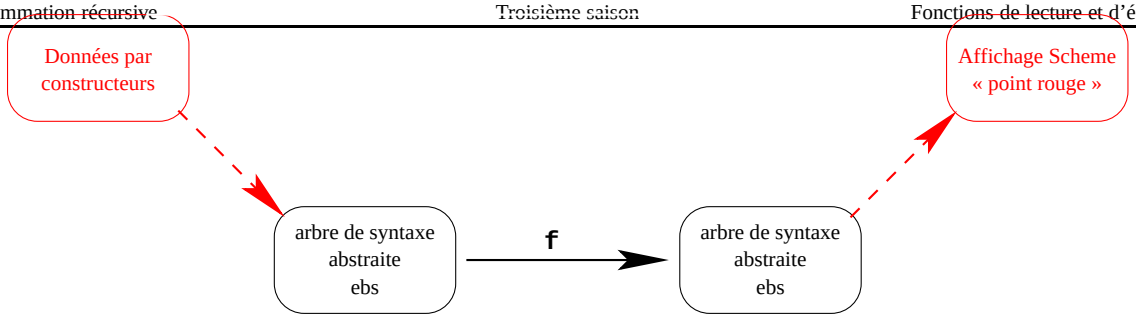
Une fonction manipulant les expressions booléennes simples peut avoir comme donnée ou comme résultat une expression booléenne simple :

- Lorsque la fonction rend une expression booléenne simple, au moins pour la tester, il faut pouvoir afficher une telle expression (rappelons que dans l'implantation que nous vous fournissons, toutes les expressions sont affichées avec un point rouge).
- Lorsque la fonction a comme donnée une expression booléenne simple, jusqu'à présent, pour la tester, nous avons « fabriqué » une telle expression booléenne en utilisant les constructeurs de la barrière syntaxique. Or, l'intérêt d'avoir un langage est d'utiliser des mots du langage comme données à de telles fonctions.

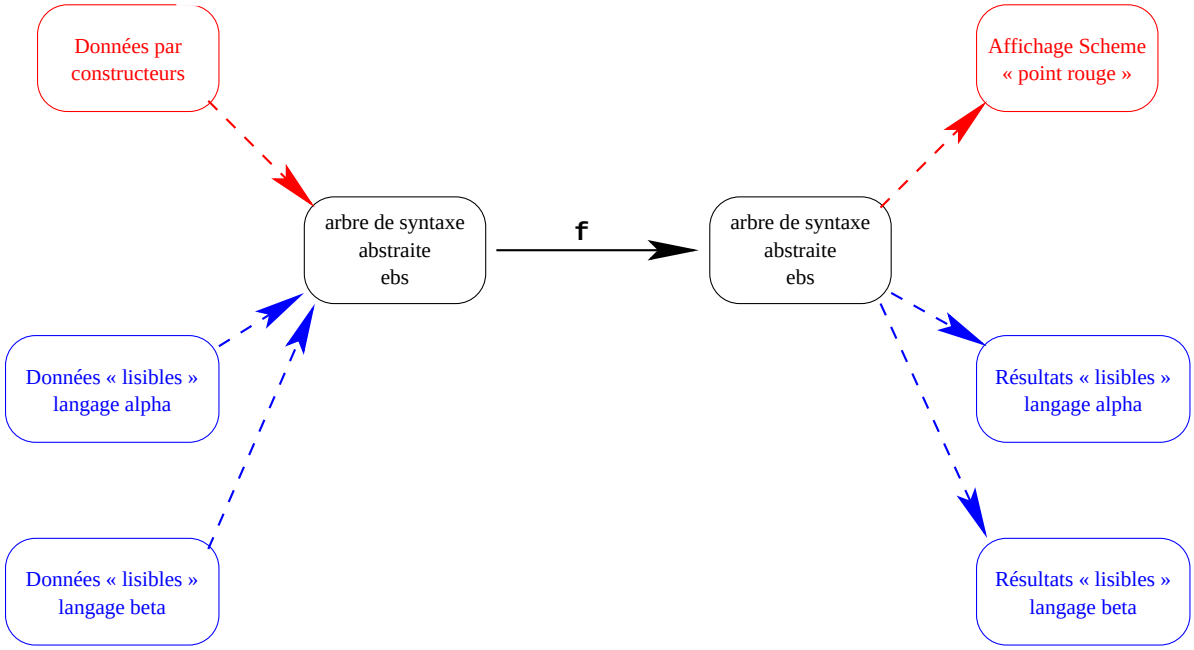
Considérons le cas d'une fonction, `f`, qui regroupe tous les problèmes, à savoir que la donnée et le résultat de la fonction sont des expressions booléennes simples :



Pour tester cette fonction, actuellement, travaillant uniquement avec la barrière syntaxique, la donnée doit être fournie uniquement à l'aide des constructeurs et l'affichage est, systématiquement, un point rouge (il faut être un expert plus que confirmé pour savoir si c'est le bon résultat...) :



Le but de cette section est d'écrire différentes fonctions de conversion pour pouvoir
- dans les programmes, écrire les données dans un langage lisible par l'homme (et comme les données sont des expressions booléennes simples, nous utiliserons des langages des expressions booléennes simples),
- convertir, pour que ce soit lisible par l'homme, des expressions booléennes simples dans un des langages des expressions booléennes simples :



Dans un premier temps, nous regardons comment convertir une expression booléenne simple en une expression lisible par l'homme (c'est facile, nous aurions pu le laisser en exercice). Dans un deuxième temps, nous regarderons la conversion d'une donnée lisible par l'homme en une expression booléenne simple.

4.2. Fonctions de conversion de sortie

La conversion est particulièrement simple si on se contente d'une forme complètement parenthésée car on a alors une Sexpression, structure que Scheme affiche parfaitement. Comme exercice, vous pouvez écrire les différentes fonctions (en préfixé, infixé et suffixé). Nous donnons ci-dessous une définition de la fonction qui convertit l'expression en préfixé (complètement parenthésée).

4.1.1. Conversion en préfixé (complètement parenthésée)

Nous nommons cette fonction `ebs->sexpr-pref` car sa donnée est une expression booléenne simple (d'où `ebs->`) et que son résultat est une Sexpression (d'où `sexpr`) représentant l'expression booléenne simple en préfixé (d'où `pref`).

Spécification

Sa spécification est :

```
;;; ebs->sexpr-pref : ExprBoolSimple -> Sexpression
;;; (ebs->sexpr-pref exp) rend la Sexpression représentant exp en
;;; notation préfixée, complètement parenthésée.
```

Exemple d'application

```
(let ((expression
       (ebs-binaire (ebs-unaire (ebs-operateur1-non)
                                (ebs-binaire (ebs-constante-vrai)
                                             (ebs-operateur2-et)
                                             (ebs-constante-vrai)))
                    (ebs-operateur2-et)
                    (ebs-binaire (ebs-constante-vrai)
                                 (ebs-operateur2-ou)
                                 (ebs-constante-faux)))))
  (ebs->sexpr-pref expression))
  →(@et (@non (@et @v @v)) (@ou @v @f))
```

Implantation

Comme pour toutes les fonctions dont la donnée est un mot du langage, la forme générale de la définition de `ebs->sexpr-pref` est une conditionnelle qui passe en revue toutes les branches de la règle donnée pour l'axiome (rappelons que c'est la première règle) :

```
(define (ebs->sexpr-pref exp)
  (cond ((ebs-atomique? exp)
         ... à faire
        ((ebs-unaire? exp)
         ... à faire
         ... à faire
        ((ebs-binaire? exp)
         ... à faire
         ... à faire
         ... à faire
        (else (erreur 'ebs->sexpr-pref
                      "expression mal formée"))))
```

et il n'y a plus qu'à remplir les trous. Lorsque l'expression est réduite à une constante, la forme préfixe est l'expression elle-même :

```
(define (ebs->sexpr-pref exp)
  (cond ((ebs-atomique? exp)
         exp)
        ((ebs-unaire? exp)
         ... à faire
         ... à faire
        ((ebs-binaire? exp)
         ... à faire
         ... à faire
```

```
        (else (erreur 'ebs->sexpr-pref
                      "expression mal formée"))))
```

lorsque l'expression donnée est une expression unaire, sa forme préfixe est obtenue en écrivant l'opérateur unaire suivi de la forme préfixe de la sous-expression, le tout entre parenthèses (qui sont données par la liste) :

```
(define (ebs->sexpr-pref exp)
  (cond ((ebs-atomique? exp)
         exp)
        ((ebs-unaire? exp)
         (list (ebs-unaire-operateur exp)
               (ebs->sexpr-pref (ebs-unaire-operande exp))))
        ((ebs-binaire? exp)
         ... à faire
         ... à faire
         ... à faire
        (else (erreur 'ebs->sexpr-pref
                      "expression mal formée"))))
```

lorsque l'expression donnée est une expression binaire, sa forme préfixe est obtenue en écrivant l'opérateur binaire suivi de la forme préfixe de la sous-expression gauche suivi de la forme préfixe de la sous-expression droite, le tout entre parenthèses (qui sont données par la liste) :

```
(define (ebs->sexpr-pref exp)
  (cond ((ebs-atomique? exp)
         exp)
        ((ebs-unaire? exp)
         (list (ebs-unaire-operateur exp)
               (ebs->sexpr-pref (ebs-unaire-operande exp))))
        ((ebs-binaire? exp)
         (list (ebs-binaire-operateur exp)
               (ebs->sexpr-pref (ebs-binaire-operande-gauche exp))
               (ebs->sexpr-pref (ebs-binaire-operande-droit exp))))
        (else (erreur 'ebs->sexpr-pref
                      "expression mal formée"))))
```

4.1.2. Conversion en polonaise préfixé

Une expression en polonaise préfixé n'étant pas parenthésée, le résultat de la fonction permettant l'affichage n'est pas une Sexpression, mais une chaîne de caractères et nous nommerons `ebs->string-pol-pref` cette fonction.

Spécification

Sa spécification est :

```
;;; ebs->string-pol-pref : ExprBoolSimple -> Sexpression
;;; (ebs->string-pol-pref exp) rend la Sexpression représentant exp en
;;; notation préfixée, complètement parenthésée.
```

Exemple d'application

```
(let ((expression
       (ebs-binaire (ebs-unaire (ebs-operateur1-non)
                                (ebs-binaire (ebs-constante-vrai)
                                             (ebs-operateur2-et)
                                             (ebs-constante-vrai)))
                    (ebs-operateur2-et)
                    (ebs-binaire (ebs-constante-vrai)
                                 (ebs-operateur2-ou)
                                 (ebs-constante-faux)))))
  (ebs->string-pol-pref expression))
  → "@et @non @et @v @v @ou @v @f"
```

Implantation

Tout d'abord, notez que, dans la notation polonaise, comme nous pouvons écrire les variables avec plusieurs caractères, les différents éléments doivent être isolés. En effet, par exemple, $\vee abc$ (ou `@ouabc`) est ambigu (est-ce $\vee a\ bc$ ou $\vee ab\ c$?).

L'implantation ne pose pas de problème :

```
;;; ebs->string-pol-pref : ExprBoolSimple -> Sexpression
;;; (ebs->string-pol-pref exp) rend la Sexpression représentant exp en
;;; notation préfixée, complètement parenthésée.
(define (ebs->string-pol-pref exp)
  (cond ((ebs-atomique? exp)
         (symbol->string exp))
        ((ebs-unaire? exp)
         (string-append (symbol->string (ebs-unaire-operateur exp))
                        " "
                        (ebs->string-pol-pref (ebs-unaire-operande exp))))
        ((ebs-binaire? exp)
         (string-append
          (symbol->string  (ebs-binaire-operateur exp))
          " "
          (ebs->string-pol-pref (ebs-binaire-operande-gauche exp))
          " "
          (ebs->string-pol-pref (ebs-binaire-operande-droit exp))))
        (else (erreur 'ebs->string-pol-pref
                      "expression mal formée"))))
```

4.2. Fonctions de conversion d'entrée

Considérons, par exemple, la fonction `ebs-exp-cste?` qui prend comme donnée un élément du type `ExprBoolSimple`. Si nous voulons faire évaluer une expression d'un langage $expBoolSimple_\alpha$, nous devons avoir une fonction, α`->ebs`, qui a comme donnée un mot de $expBoolSimple_\alpha$ et qui rend une valeur du type `ExpBoolSimple` (*i.e.* un élément de la barrière syntaxique des expressions booléennes simples) :

```
;;; α->ebs: α -> ExpBoolSimple
;;; (α->ebs d) rend l'élément de ExpBoolSimple représenté,
;;; dans le langage expBoolSimple_α, par «d»
```

et, pour savoir si l'expression `exp-`α, écrite dans le langage $expBoolSimple_\alpha$, est une expression constante, il suffit d'écrire :

```
(ebs-exp-cste? (α->ebs exp-α))
```

Comme exemples, spécifions et implantons deux convertisseurs d'entrée pour les expressions booléennes simples.

4.2.1. sexpr-pref->ebs

Commençons par le langage préfixe complètement parenthésé. Un mot de ce langage étant complètement parenthésé, il peut être vu comme une Sexpression et le convertisseur d'entrée a donc comme type `S-Expression -> ExprBoolSimple`. Pour indiquer le type de la donnée et le genre (préfixe) de langage, nous nommerons `sexpr-pref->ebs` cette fonction dont la spécification est donc :

```
;;; sexpr-pref->ebs : Sexpression -> ExprBoolSimple
;;; (sexpr-pref->ebs s) rend l'arbre de syntaxe abstraite dont la représentation,
;;; dans le langage exprBoolSimple_prefixe est la Sexpression s
```

En voici une application :

```
(ebs-exp-cste? (sexpr-pref->ebs '(@ou (@et a @f) (@non @f))))
```

Pour implanter cette fonction, on analyse la Sexpression à lire :

- soit elle est réduite à un symbole (ce n'est pas une liste) et
  - soit c'est `@v` ou `@f` et il suffit de fabriquer l'expression booléenne représentée par cette constante,

- sinon, c'est une variable et il suffit de fabriquer l'expression booléenne représentée par cette variable,
  - soit c'est une Sexpression de longueur 2 et il suffit de fabriquer l'expression composée unaire dont
    - l'opérateur est représenté par le premier élément de la Sexpression donnée,
    - l'opérande est obtenue en lisant (appel récursif) le deuxième élément de la Sexpression donnée,
  - soit c'est une Sexpression de longueur 3 et il suffit de fabriquer l'expression composée binaire dont
    - l'opérande gauche est obtenue en lisant (appel récursif) le deuxième élément de la Sexpression donnée,
    - l'opérateur est représenté par le premier élément de la Sexpression donnée,
    - l'opérande droit est obtenue en lisant (appel récursif) le troisième élément de la Sexpression donnée.

D'où l'implantation :

```
(define (sexpr-pref->ebs s)
  (if (pair? s)
      (if (pair? (cdr s))
          (if (pair? (cddr s))
              (if (pair? (cdddr s))
                                        ; la liste a au moins 4 éléments:
                  (erreur 'sexpr-pref->ebs s " mal formée (+4)")
                                        ; la liste a 3 éléments:
                  (ebs-binaire (sexpr-pref->ebs (cadr s))
                               (sexpr-pref-operateur2 (car s))
                               (sexpr-pref->ebs (caddr s))))
                                        ; la liste a 2 éléments:
              (ebs-unaire (sexpr-pref-operateur1 (car s))
                          (sexpr-pref->ebs (cadr s))))
                                        ; la liste a 1 élément:
          (erreur 'sexpr-pref->ebs s " mal formée (1)"))
      (if (sexpr-pref-constante? s)
          (sexpr-pref-constante s)
          (sexpr-pref-variable s))))

;;; sexpr-pref-constante? : Symbole -> bool
;;; (sexpr-pref-constante? s) rend #t ssi s représente une constante
(define (sexpr-pref-constante? s)
  (member s '(@v @f)))
;;; sexpr-pref-constante : Symbole -> ExprBoolSimple
;;; (sexpr-pref-constante s) rend l'expression réduite à la constante
;;; représentée par s
(define (sexpr-pref-constante s)
  (cond ((equal? s '@v) (ebs-constante-vrai))
        ((equal? s '@f) (ebs-constante-faux))
        (else (erreur 'sexpr-pref->ebs s "n'est pas une constante"))))
;;; sexpr-pref-variable : Symbole -> ExprBoolSimple
;;; (sexpr-pref-variable s) rend l'expression réduite à la variable
;;; représentée par s
(define (sexpr-pref-variable s)
  (ebs-variable s))

;;; sexpr-pref-operateur1 : Symbole -> Operateur1
;;; (sexpr-pref-operateur1 s) rend l'opérateur unaire représenté par s
(define (sexpr-pref-operateur1 s)
  (if (equal? s '@non)
      (ebs-operateur1-non)
      (erreur 'sexpr-pref->ebs s "n'est pas un opérateur unaire")))
;;; sexpr-pref-operateur2 : Symbole -> Operateur2
;;; (sexpr-pref-operateur2 s) rend l'opérateur binaire représenté par s
(define (sexpr-pref-operateur2 s)
  (cond ((equal? s '@et) (ebs-operateur2-et))
        ((equal? s '@ou) (ebs-operateur2-ou))
        (else (erreur 'sexpr-pref->ebs s "n'est pas un opérateur binaire"))))
```

4.2.2. polonaise-prefixe->ebs

Donnons la grammaire :

<expBoolSimple$_{polonaise}$> → *<atomique>*
<unaire>
<binaire>

<unaire> → *<opérateur1>* *<expBoolSimple$_{polonaise}$>*

<binaire> → *<opérateur2>* *<expBoolSimple$_{polonaise}$>* *<expBoolSimple$_{polonaise}$>*

<atomique> → *<constante>*
<variable>

<constante> → `@f` FAUX
`@v` VRAI

<variable> → *Une suite de caractères autres que l'espace et les parenthèses*

<opérateur1> → `@non`

<opérateur2> → `@et` ET
`@ou` OU

Exemple : @et@non@ou@f@v@f

Premier problème :

L'écriture d'un convertisseur d'entrée est alors beaucoup moins simple que précédemment. En effet `@et @non @ou @f @v @f` n'est pas une Sexpression et nous avons vu qu'en Scheme seules les Sexpressions pouvaient être des données.

Il faut donc tricher. Deux solutions :

- entourer la donnée par des parenthèses (car on a alors une liste, cas particulier d'une Sexpression) ; l'application du convertisseur d'entrée est alors
  `(polonaise-prefixe->ebs '(@et @non @ou @f @v @f))`
- considérer la donnée comme une chaîne de caractères (rappelons que les chaînes de caractères sont des Sexpressions « atomiques ») ; l'application du convertisseur d'entrée est alors
  `(polonaise-prefixe->ebs "@et @non@ou @f @v @f")`

la seconde solution étant celle où l'on triche le moins, mais il semble que la première solution soit plus simple à traiter, aussi regardons cette solution.

Second problème :

Mais il y a un autre problème ! Pour analyser la liste (@et @non @ou @f @v @f), il faut la découper (on peut facilement voir qu'il s'agit d'un « binaire ») en

- son opérateur, `@et` : facile, c'est le `car` de la liste ;
- son premier opérande `(@non @ou @f @v)` : comment faire cela ? diffi cile...
- son second opérande `(@f)`...

Ainsi, même en trichant au maximum, la convertion d'une expression écrite en polonaise préfi xée est très diffi cile. Aussi nous ne le ferons pas... (bien qu'étant hors programme de ce cours, un convertisseur, ayant comme donnée une chaîne de caractères, est fourni en annexe).

Noter que la conversion des expressions écrites « normalement » est encore plus diffi cile puisque, en plus du découpage, il faut gérer les priorités des opérateurs. Pour les expressions booléennes, nous vous fournirons un convertisseur des expressions écrites « normalement » (sans vous donner le source car on s'éloigne beaucoup trop du programme de ce cours).

5. Implantation de la barrière syntaxique

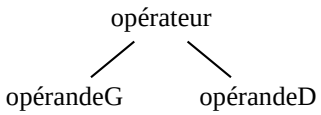
[♠♠♠ À revoir entièrement (arbres bornés 2 ? Sexpression ?) ♠♠♠]

Nous nous intéressons maintenant à l'implantation de la barrière syntaxique.

On peut implanter l'arbre de syntaxe abstraite des expressions booléennes simples à l'aide des Sexpressions :

- une constante est tout simplement implantée par le symbole Scheme correspondant,
- un opérateur est tout simplement implanté par le symbole Scheme correspondant,
- une expression unaire est implantée par une Sexpression ayant deux éléments, la Sexpression implantant l'opérateur et la Sexpression implantant l'opérande.

Une expression binaire est implantée par une Sexpression ayant trois éléments, les Sexpressions implantant, dans l'ordre, l'opérande gauche, puis l'opérateur et enfin l'opérande droit.



et nous décidons – mais nous pourrions faire un autre choix – que le premier élément de la liste correspond à l'opérande gauche (qui est une expression), que le second élément correspond à l'opérateur et que le dernier élément correspond à l'opérande droit (qui est une expression).

⇒ `(opérandeG opérateur opérandeD)`

Remarque : il faut bien comprendre que ceci n'est pas un arbre borné-deux puisque les deux sous-arbres et la racine sont mis sur le même plan : ce n'est qu'une implantation, parmi d'autres, de l'arbre de syntaxe abstraite.

Les définitions des différentes fonctions de la barrière syntaxique s'écrivent alors facilement (nous ne l'étudierons donc pas, nous vous le donnons en annexe).

5.1. sexpr-infixe->ebs

Et maintenant, transgressons nos règles ! ! !

Pour savoir si l'expression booléenne simple `exp` écrite en infixé, complètement parenthésée, est une expression constante, en utilisant la forme générale que nous avons donnée page 19, nous écririons :

```
(ebs-exp-cste? (sexpr-infixe->ebs exp))
```

Mais, rappelons que nous avons implanté l'arbre de syntaxe abstraite avec une Sexpression en prenant, dans le cas d'une expression binaire, comme premier élément l'opérande gauche, comme deuxième élément l'opérateur et comme troisième élément l'opérande droit. Considérons alors le langage $expBoolSimple_{infixe}$: quelle que soit sa donnée, la fonction `sexpr-infixe->ebs` rend la donnée elle-même (on dit que c'est la fonction *identité*).

Pour savoir si l'expression booléenne simple `exp`, écrite dans le langage $expression_{infixe}$, il suffit donc d'écrire, en traversant la barrière syntaxique :

```
(ebs-exp-cste? exp)
```

Par exemple :

```
;;; exemple d'application de ebs-exp-cste?:
(ebs-exp-cste? '((@non (@v @et @v)) @ou @f)) →#t
```

De même, pour afficher une expression en infixé, complètement parenthésée, il suffit de faire afficher, en traversant encore la barrière syntaxique, par l'interprète Scheme, la valeur de l'expression dans l'implantation infixe de la barrière syntaxique.

5.2. Retour sur les expressions en préfixé

Dans les paragraphes précédents, nous avons considéré que l'on voulait utiliser plusieurs sortes d'expressions (en infixe, suffixe...) en même temps. Dans la réalité, le plus souvent, on n'utilise qu'une notation et on s'y tient (au moins un certain temps...). Nous avons vu aussi qu'il était alors plus simple d'avoir une implantation de la barrière syntaxique en accord avec la notation retenue. Ainsi, dans l'exemple, nous avions décidé de prendre une notation infixe, aussi avons-nous implanté la barrière syntaxique en utilisant des Sexpressions où l'opérateur est en position centrale. Mais on peut changer d'avis... Si, après avoir décidé d'utiliser une notation infixe, nous décidons de n'utiliser que des expressions préfixe, au lieu d'écrire des convertisseurs d'entrée et de sortie, il y a plus simple, plus sûr et plus efficace : changer l'implantation de la barrière d'abstraction en utilisant une implantation préfixe (et, ainsi, les convertisseurs deviennent inutiles). Voici le source des seules définitions à modifier (la définiton de `ebs-binaire-operande-droite` restant la même par hazard) :

```
;;; ebs-binaire-operande-gauche : Binaire -> ExprBoolSimple
;;; (ebs-binaire-operande-gauche exp) rend l'expression, opérande gauche de
;;; l'expression donnée
(define (ebs-binaire-operande-gauche exp)
```

```
  (car exp))

;;; ebs-binaire-operateur : Binaire -> Operateur2
;;; (ebs-binaire-operateur exp) rend l'opérateur (principal) de l'expression donnée
(define (ebs-binaire-operateur exp)
  (car exp))

;;; ebs-binaire : ExprBoolSimple * Operateur2 * ExprBoolSimple -> ExprBoolSimple
;;; (ebs-binaire exp1 op exp2) rend l'expression composée ayant comme
;;; opérande gauche exp1, comme opérateur op et comme opérande droit exp2
(define (ebs-binaire exp1 op exp2)
  (list op exp1 exp2))
```

5.3. Conclusion

Aussi, dans la plupart des cas, nous tricherons :

- en prenant un langage complètement parenthésée (*i.e.* une Sexpression),
- en implantant l'arbre de syntaxe abstraite avec la même forme que le langage considéré,
- et si nous voulons changer de forme de représentation (infixe, préfixe, postfixe...), nous modifierons carrément l'implantation de la barrière d'abstraction.

6. Barrière d'interprétation

6.1. Barrières d'interprétation

Si l'on veut évaluer les expressions, encore faut-il savoir à quoi correspondent les terminaux. Ainsi, dans notre exemple, il faut savoir que les symboles `@v` et `@f` correspondent aux valeurs de vérité – mais quelles valeurs de vérité ? – et que les symboles `@ou`, `@et`, et `@non` correspondent respectivement à la disjonction, la conjonction et la négation.

Ainsi doit-on utiliser cinq fonctions, qui constituent la barrière d'interprétation du langage :

```
;;; faux :  -> Booleen
;;; (faux) rend la valeur de vérité « faux »


;;; vrai :  -> Booleen
;;; (vrai) rend la valeur de vérité « vrai »


;;; non : Booleen -> Booleen
;;; (non b) rend la négation de b dans le type Booleen


;;; et : Booleen * Booleen -> Booleen
;;; (et b1 b2) rend la conjonction de b1 et de b2 dans le type Booleen


;;; ou : Booleen * Booleen -> Booleen
;;; (ou b1 b2) rend la disjonction de b1 et de b2 dans le type Booleen
```

Implantations de la barrière d'interprétation

Donnons une première implantation, dans les booléens Scheme :

```
;;;; Interprétation avec Booleen = #f, #t
(define (faux)
  #f)
(define (vrai)
```

```
#f)
(define (faux? b)
  (not b))
(define (vrai? b)
  b)
(define (non b)
  (not b))
(define (et b1 b2)
  (and b1 b2))
(define (ou b1 b2)
  (or b1 b2))
```

Noter l'implantation des opérations qui sont des fonctions qui retournent une fonction.
Une autre implantation, en prenant comme valeurs booléennes les symboles `@v` et `@f` :

```
;;;; Interprétation avec Booleen = @f, @v
(define (faux) '@f)

(define (vrai) '@v)

(define (faux? b)
  (equal? b '@f))

(define (vrai? b)
  (equal? b '@v))

(define (non b)
  (if (equal? b '@f) '@v '@f))

(define (et b1 b2)
  (if (and (equal? b1 '@v) (equal? b2 '@v)) '@v '@f))

(define (ou b1 b2)
  (if (or (equal? b1 '@v) (equal? b2 '@v)) '@v '@f))
```

6.2. Évaluation des expressions constantes

Nous avons spécifié toutes les fonctions nécessaires à l'écriture de la fonction `ebs-eval-expr-cste` qui a comme spécification :

```
;;; ebs-eval-expr-cste : ExprBoolSimple -> Booleen
;;; (ebs-eval-expr-cste exp) rend la valeur de vérité de l'expression exp
```

En effet, voici une définition de cette fonction :

```
(define (ebs-eval-expr-cste exp)
  (cond ((ebs-constante? exp)
         (ebs-eval-constante exp))
        ((ebs-unaire? exp)
         (ebs-eval-unaire exp))
        ((ebs-binaire? exp)
         (ebs-eval-binaire exp))
        (else (erreur 'ebs-eval-cste "expression mal formée"))))
```

```
;;; ebs-eval-constante : Constante -> Booleen
;;; (ebs-eval-constante cste) rend la valeur de vérité de la constante cste
(define (ebs-eval-constante cste)
  (if (ebs-constante-vrai? cste)
```

```
      (vrai)
      (faux)))
;;; ebs-eval-unaire : Unaire -> Booleen
;;; (ebs-eval-unaire exp) rend la valeur de vérité de l'expression unaire exp
(define (ebs-eval-unaire exp)
  (let ((operateur (ebs-unaire-operateur exp))
        (operande (ebs-unaire-operande exp)))
    (if (ebs-operateur1-non? operateur)
        (non (ebs-eval-expr-cste operande))
        (erreur 'ebs-eval-unaire operateur "n'est pas un opérateur unaire"))))

;;; ebs-eval-binaire : Binaire -> Booleen
;;; (ebs-eval-binaire exp) rend la valeur de vérité de l'expression binaire exp
(define (ebs-eval-binaire exp)
  (let ((operateur (ebs-binaire-operateur exp))
        (operandeG (ebs-binaire-operande-gauche exp))
        (operandeD (ebs-binaire-operande-droit exp)))
    (cond ((ebs-operateur2-et? operateur)
           (et
            (ebs-eval-expr-cste operandeG)
            (ebs-eval-expr-cste operandeD)))
          ((ebs-operateur2-ou? operateur)
           (ou
            (ebs-eval-expr-cste operandeG)
            (ebs-eval-expr-cste operandeD)))
          (else (erreur 'ebs-eval-binaire operateur
                        "n'est pas un opérateur binaire")))))
```

Remarque très importante : noter que cette définition est valable quel que soit le langage utilisé (préfixe, infixe...).

7. Transformations d'expressions booléennes simples

Nous voudrions écrire la fonction `ebs-simplifiee` qui évalue partiellement une expression booléenne simple en éliminant, au maximum, les constantes booléennes.

7.1. Spécification

Ainsi cette fonction a comme donnée une expression booléenne simple (avec variables) et comme résultat une autre expression, simplifiée de l'expression donnée :

```
;;; ebs-simplifiee : ExprBoolSimple -> ExprBoolSimple
;;;  ERREUR lorsque exp n'est pas une expression bien formée
;;; (ebs-simplifiee expr) rend une expression booléenne simple simplifiée
;;; équivalente à l'expression booléenne simple donnée.
```

Exemple :

```
(ebs-simplifiee '((@non a) @et (b @ou (@v @et a))))
→ ((@non a) @et (b @ou a))
(ebs-simplifiee '((@non @f) @et (b @ou (@v @et a))))
→ (b @ou a)
(ebs-simplifiee '((@non @f) @et (b @ou (@v @ou a))))
→ @v
```

Règles de simplification :

Encore faut-il préciser ce que nous entendons par « simplifiée ». Nous utiliserons les simplifications suivantes :

a (@ou @v) ⇒ @v	(a @et @v) ⇒ a	(non @v) ⇒ @f
(a @ou @f) ⇒ a	(a @et @f) ⇒ @f	(non @f) ⇒ @v
(@v @ou a) ⇒ @v	(@v @et a) ⇒ a	
(@f @ou a) ⇒ a	(@f @et a) ⇒ @f	

Remarques :

1. il s'agit bien d'une évaluation partielle, d'autant plus que les sous-expressions qui n'ont pas d'occurrences de variables sont évaluées ;
2. nous pourrions donner d'autres règles de simplification (par exemple `(@non (@non a))` ⇒`a`). Vous pouvez essayer d'implanter cette dernière règle de simplification – et, éventuellement, d'autres. Nous nous en tiendrons là.

7.2. Implantation

Encore le même schéma :

```
(define (ebs-simplifiee exp)
  (cond ((ebs-atomique? exp)

         ((ebs-unaire? exp)


         ((ebs-binaire? exp)



         (else (erreur 'ebs-simplifiee "expression mal formée"))))
```

une formule atomique est simplifiée ; on la retourne donc telle quelle :

```
(define (ebs-simplifiee exp)
  (cond ((ebs-atomique? exp)
         exp)
         ((ebs-unaire? exp)


         ((ebs-binaire? exp)



         (else (erreur 'ebs-simplifiee "expression mal formée"))))
```

Pour les expressions unaires et binaires, cela semble plus compliqué : nous utilisons des fonctions `ebs-unaire-simplifiee` et `ebs-binaire-simplifiee` qui simplifient des expressions unaires et binaires, mais données par leurs décompositions :

```
(define (ebs-simplifiee exp)
  (cond ((ebs-atomique? exp)
         exp)
         ((ebs-unaire? exp)
          (ebs-unaire-simplifiee (ebs-unaire-operateur exp)
                                 (ebs-unaire-operande exp)))
         ((ebs-binaire? exp)
          (ebs-binaire-simplifiee (ebs-binaire-operateur exp)
                                  (ebs-binaire-operande-gauche exp)
                                  (ebs-binaire-operande-droit exp)))
         (else (erreur 'ebs-simplifiee "expression mal formée"))))
```

avec ces fonctions spécifiées par :

```
;;; ebs-unaire-simplifiee : Operateur1 * ExprBoolSimple -> ExprBoolSimple
;;; (ebs-unaire-simplifiee op exp) rend une expression booléenne simple
```

```
;;; simplifiée équivalente à l'expression booléenne (ebs-unaire op exp).
```

```
;;; ebs-binaire-simplifiee : Operateur2 * ExprBoolSimple * ExprBoolSimple -> ExprBoolSimple
;;; (ebs-binaire-simplifiee op expG expD) rend une expression booléenne
;;; simple simplifiée équivalente à l'expression booléenne
;;; (ebs-binaire expG op expD).
```

7.2.1. Implantation de `ebs-unaire-simplifiee`

Pour la simplification d'une expression unaire, on vérifie que l'opérateur est bien `@non` et, si c'est bien le cas, on fait appel à la fonction de simplification dédiée à la négation. En remarquant que la simplification d'une négation doit être calculée à partir de la simplification de sa sous-expression :

```
;;; ebs-unaire-simplifiee : Operateur1 * ExprBoolSimple -> ExprBoolSimple
;;; (ebs-unaire-simplifiee op exp) rend une expression booléenne simple
;;; simplifiée équivalente à l'expression booléenne (ebs-unaire op exp).
(define (ebs-unaire-simplifiee op exp)
  (if (ebs-operateur1-non? op)
      (ebs-neg-simplifiee (ebs-simplifiee exp))
      (erreur 'ebs-simplifiee op "n'est pas un opérateur unaire")))
```

en utilisant une fonction, `ebs-neg-simplifiee` qui doit avoir comme spécification :

```
;;; ebs-neg-simplifiee : ExprBoolSimple/simplifiee/ -> ExprBoolSimple/simplifiee/
;;; (ebs-neg-simplifiee exp) rend une expession booléenne simple simplifiée
;;; équivalente à (non exp)
```

7.2.2. Implantation de `ebs-binaire-simplifiee`

De même, pour simplifier une expression binaire, on fait appel à une fonction de simplification dédiée à l'opérateur de l'expression :

```
(define (ebs-binaire-simplifiee op expG expD)
  (cond ((ebs-operateur2-et? op)
         (ebs-conj-simplifiee (ebs-simplifiee expG) (ebs-simplifiee expD)))
        ((ebs-operateur2-ou? op)
         (ebs-disj-simplifiee (ebs-simplifiee expG) (ebs-simplifiee expD)))
        (else (erreur 'ebs-simplifiee op "n'est pas un opérateur binaire"))))
```

ces fonctions ayant comme spécification :

```
;;; ebs-conj-simplifiee : ExprBoolSimple/simplifiee/ * ExprBoolSimple/simplifiee/
;;;   -> ExprBoolSimple/simplifiee/
;;; (ebs-conj-simplifiee e1 e2) rend une expession booléenne simple simplifiée
;;; équivalente à (e1 et e2)
```

```
;;; ebs-disj-simplifiee : ExprBoolSimple/simplifiee/ * ExprBoolSimple/simplifiee/
;;;   -> ExprBoolSimple/simplifiee/
;;; (ebs-disj-simplifiee e1 e2) rend une expession booléenne simple simplifiée
;;; équivalente à (e1 ou e2)
```

7.2.3. Implantation de `ebs-neg-simplifiee`

Règles de simplification (rappel) :

Rappelons les règles de simplification pour la négation :

`(@non @v)` $\Rightarrow$ `@f`
`(@non @f)` $\Rightarrow$ `@v`

ce à quoi on pourrait ajouter, pour tenir compte des cas où il n'y a pas de simplification :

`(@non a)` $\Rightarrow$ `(@non a)`

Rappelons la spécification de la fonction `ebs-neg-simplifiee` :

```
;;; ebs-neg-simplifiee : ExprBoolSimple/simplifiee/ -> ExprBoolSimple/simplifiee/
;;; (ebs-neg-simplifiee exp) rend une expession booléenne simple simplifiée
;;; équivalente à (non exp)
```

et notons que la donnée de cette fonction est une expression simplifiée et que c'est la sous-expression de la négation (c'est le `a` de la règle ci-dessus). L'implantation est alors facile (noter tout de même l'utilisation de la barrière syntaxique) :

```
(define (ebs-neg-simplifiee exp)
  (cond ((ebs-constante-vrai? exp) (ebs-constante-faux))
        ((ebs-constante-faux? exp) (ebs-constante-vrai))
        (else  (ebs-unaire (ebs-operateur1-non) exp))))
```

7.2.4. Implantation de `ebs-conj-simplifiee`

Règles de simplification (rappel) :

Rappelons les règles de simplification pour la conjonction :

`(@f @et a)` $\Rightarrow$ `@f`	`(a @et @f)` $\Rightarrow$ `@f`
`(@v @et a)` $\Rightarrow$ `a`	`(a @et @v)` $\Rightarrow$ `a`

en se rappelant que la donnée de la fonction `ebs-conj-simplifiee` est constituée par les deux sous-expressions simplifiée d'une conjonction, l'implantation de cette fonction est :

```
;;; ebs-conj-simplifiee : ExprBoolSimple/simplifiee/ * ExprBoolSimple/simplifiee/
;;;   -> ExprBoolSimple/simplifiee/
;;; (ebs-conj-simplifiee e1 e2) rend une expession booléenne simple simplifiée
;;; équivalente à (e1 et e2)
(define (ebs-conj-simplifiee e1 e2)
  (cond ((ebs-constante-faux? e1) (ebs-constante-faux))
        ((ebs-constante-faux? e2) (ebs-constante-faux))
        ((ebs-constante-vrai? e1) e2)
        ((ebs-constante-vrai? e2) e1)
        (else  (ebs-binaire e1 (ebs-operateur2-et) e2))))
```

Idem pour `ebs-disj-simplifiee` qui est laissée en exercice.

8. Notion d'environnement

Nous voudrions évaluer les expressions booléennes simples ayant des variables. Mais pour ce faire, il faut associer une valeur à chaque variable. Autrement dit, le problème est d'évaluer une expression booléenne bien formée dans un **environnement** qui est représenté par une liste d'associations `(clef valeur)`. Par exemple,

env : `a` $\rightsquigarrow$ `@v`
`b` $\rightsquigarrow$ `@f`

si, dans `env`, la valeur de `a` est `@v` et la valeur de `b` est `@f` :

`(ebs-eval '((@non a) @et (b @ou (@v @et a))) env)`$\rightarrow$`@f`

8.0.5. Spécification de `ebs-eval`

```
;;; ebs-eval : ExprBoolSimple * Environnement -> Booleen
;;; (ebs-eval exp env) rend la valeur de vérité de l'expression exp dans
```

;;; *l'environnement env*

8.0.6. Création d'un environnement

Il faut que l'on puisse créer un environnement qui associe à chaque variable une valeur. Pour ce faire, nous pourrions définir une fonction qui aurait comme donnée une liste d'associations variable — valeur et qui créerait l'environnement voulu. Mais, la plupart du temps, lorsqu'on travaille sous un environnement, au départ, on ne part pas de zéro, mais, au contraire, dans un environnement initial. Aussi, pour créer des environnements, préfère-t-on avoir deux fonctions : la première – `env-initial` – rend l'environnement initial et la seconde – `env-ajouts` – permet de rajouter des associations variable — valeur. Ainsi, l'exemple précédent s'écrira :

```
(let((env (env-ajouts (list (list 'a (vrai))
                            (list 'b (faux)))
                      (env-initial))))
  (ebs-eval '((@non a) @et (b @ou (@v @et a))) env))
```

8.0.7. Implantation de `ebs-eval`

La définition de la fonction `ebs-eval` suit toujours le même schéma. Nous ne le détaillons pas :

```
(define (ebs-eval exp env)
  (cond ((ebs-atomique? exp)
         (ebs-eval-atomique exp env))
        ((ebs-unaire? exp)
         (ebs-eval-unaire exp env))
        ((ebs-binaire? exp)
         (ebs-eval-binaire exp env))
        (else (erreur 'ebs-eval "expression mal formée"))))
```

Les fonctions `non`, `et` et `ou` sont des fonctions de la barrière d'interprétation que nous avons déjà utilisées pour évaluer une expression constante. Reste la fonction `atomique-val` qui doit avoir comme spécification :

```
;;; ebs-eval-atomique : Atomique * Environnement -> Booleen
;;; (ebs-eval-atomique exp env) rend la valeur de vérité de l'expression
;;; atomique exp dans l'environnement env
```

8.0.8. Première implantation de `ebs-eval-atomique`

Il suffit, une fois de plus, de suivre la grammaire : une expression atomique est une constante ou une variable :

```
;;; ebs-eval-atomique : Atomique * Environnement -> Booleen
;;; (ebs-eval-atomique exp env) rend la valeur de vérité de l'expression
;;; atomique exp dans l'environnement env
(define (ebs-eval-atomique exp env)
  (if (ebs-constante? exp)
      (if (ebs-constante-vrai? exp)
          (vrai)
          (faux))
      (env-variable-val exp env)))
```

et

- pour une constante, c'est soit la constante vraie, soit la constante faux et il suffit d'utiliser les fonctions (de la barrière d'interprétation) `vrai` et `faux`,
- pour une variable, nous rajoutons, dans la barrière d'abstraction des environnements, la fonction `env-variable-val` de spécification :

```
;;; env-variable-val : Variable * Environnement[Variable Domaine] -> Domaine
;;; ERREUR lorsque exp n'est pas définie dans l'environnement
;;; (env-variable-val exp env) rend la valeur de exp dans l'environnement env
```

Barrière d'abstraction des environnements

Rappelons les différentes fonctions que nous avons mises sur les environnements en remarquant que l'environnement initial est vide :

```
;;; env-initial : -> Environnement[Variable Domaine]
;;; (env-initial) rend l'environnement vide.
```

```
;;; env-ajouts : LISTE[N-UPLET[Variable Domaine]] * Environnement[Variable Domaine]
;;;              -> Environnement[Variable Domaine]
;;; (env-ajouts L env) rend l'environnement obtenu en étendant
;;; l'environnement env en associant à chaque variable de la liste
;;; d'associations L, la valeur correspondante.
```

```
;;; env-variable-val : Variable * Environnement[Variable Domaine] -> Domaine
;;; ERREUR lorsque exp n'est pas défi nie dans l'environnement
;;; (env-variable-val exp env) rend la valeur de exp dans l'environnement env
```

Implantation de la barrière d'abstraction des environnements

Nous n'étudierons pas l'implantation qui est facile (en fait, nous avons déjà vu de telles implantations lorsque nous avons étudié les listes d'associations) ; elle vous est donnée en annexe.

8.0.9. Seconde implantation de `ebs-eval-atomique`

En fait, il s'agit plutôt d'une seconde vision de l'environnement : au lieu de tester si l'expression atomique est une constante ou une variable, on peut mettre la valeur des constantes dans l'environnement initial et le calcul de `ebs-eval-atomique` est alors tout simplement le calcul de `env-variable-val` de la solution précédente. Ainsi, la fonction `ebs-eval-atomique` n'existe plus et la fonction `ebs-eval-atomique`, renommée `env-atomique-val`, est une fonction de la barrière d'abstraction des environnements.

Barrière d'abstraction des environnements

La barrière d'abstraction des environnements est donc :

```
;;; env-initial : -> Environnement[Symbole Booleen]
;;; (env-initial) rend l'environnement associant aux deux constantes
;;; booléennes leurs valeurs dans Booleen
```

```
;;; env-ajouts : LISTE[N-UPLET[Variable Domaine]] * Environnement[Symbole Domaine]
;;;              -> Environnement[Symbole Domaine]
;;; (env-ajouts L env) rend l'environnement obtenu en étendant
;;; l'environnement env en associant à chaque variable de la liste
;;; d'associations L, la valeur correspondante.
```

```
;;; env-atomique-val : Symbole * Environnement[Symbole Domaine] -> Domaine
;;; ERREUR lorsque exp n'est pas défi nie dans l'environnement
;;; (env-atomique-val exp env) rend la valeur de exp dans l'environnement env
```

Implantation de la barrière d'abstraction des environnements

Là encore, l'implantation est facile et elle vous est donnée dans l'annexe.

8.0.10. Différence sémantique entre ces deux visions

Considérons l'environnement `env` défi ni comme suit :

```
(env-ajouts (list (list '@v (faux)))
            (env-initial))
```

— Quelle est la valeur retournée, dans les deux implantations précédentes, lorsque l'on évalue l'expression définie à la constante `@v` dans cet environnement ?

Première vision (environnement initial vide)

```
(let ((env (env-ajouts (list (list '@v (faux)))
                       (env-initial))))
  (ebs-eval '@v env))→@v
```

Seconde vision (valeurs constantes dans environnement initial)

```
(let ((env (env-ajouts (list (list '@v (faux)))
                       (env-initial))))
  (ebs-eval '@v env))→@f
```

Constante — variable prédéfinie

En fait, dans la première vision, `@v` et `@f` sont des constantes (leur valeur ne change jamais) alors que dans la seconde vision, `@v` et `@f` sont traités comme des variables sauf que, dès le départ, elles ont une valeur : on dit que ce sont des variables prédéfi nies.

Ces notions de constantes et de variables prédéfi nies se retrouvent dans tous les langages (sous des noms différents, en particulier avec la notion de mot-clef qui correspond à nos constantes. Par exemple, en Scheme, on peut utiliser n'importe quel symbole comme nom de fonction, sauf `and`, `or`, `define`... Ainsi, on peut redéfi nir `+` (qui est bien sûr prédéfi ni), mais on ne peut pas redéfi nir `and`.

9. Expressions booléennes généralisées

9.1. Grammaire

On défi nit les expressions booléennes généralisées par la grammaire :

<expBoolGen> → *<atomique>*
<négation>
<n-aire>

<atomique> → *<constante>*
<variable>

<constante> → `@v` VRAI
`@f` FAUX

<variable> → *Une suite de caractères autres que l'espace, les parenthèses et @*

<négation> → `(@non` *<expBoolGen>* `)`

<n-aire> → `(@et` *<expBoolGen>**`)` CONJONCTION
`(@ou` *<expBoolGen>**`)` DISJONCTION

Remarque : comme en Scheme, les disjonctions et les conjonctions opèrent sur un nombre quelconque d'arguments, y compris 1 argument ou même 0 argument.

9.1.1. Exemples

```
(@et (@non (@et @v a)) (@ou b (@et @v a)))
(@ou a)
(@ou)
```

Lorsqu'il y a un argument, la sémantique va de soi : la valeur de vérité d'une conjonction ou d'une disjonction n'ayant qu'une sous-expression est égale à la valeur de vérité de la sous-expression. La valeur de vérité d'une disjonction sans argument est « faux » et la valeur de vérité d'une conjonction sans argument est « vrai ».

9.2. Barrières d'abstraction

9.2.1. Barrière syntaxique

```
;;;; Reconnaisseurs :
;;; ebg-atomique? : ExprBoolGen -> bool
;;; ebg-constante? : ExprBoolGen -> bool
;;; ebg-variable? : Atomique -> bool
;;; ebg-constante-vrai? : Constante -> bool
;;; ebg-constante-faux? : Constante -> bool
;;; ebg-negation? : ExprBoolGen -> bool
;;; ebg-conjonction? : ExprBoolGen -> bool
;;; ebg-disjonction? : ExprBoolGen -> bool

;;;; Accesseurs :
;;; ebg-neg-sous-exp : Negation -> ExprBoolGen
;;; ebg-n-aire-sous-exps : N-aire -> LISTE[ExprBoolGen]

;;;; Constructeurs :
;;; ebg-constante-vrai : -> Constante
;;; ebg-constante-faux : -> Constante
;;; ebg-variable : Symbole -> ExprBoolGen
;;; ebg-negation : ExprBoolGen -> ExprBoolGen
;;; ebg-conjonction : LISTE[ExprBoolGen] -> ExprBoolGen
;;; ebg-disjonction : LISTE[ExprBoolGen] -> ExprBoolGen
```

9.2.2. Barrière d'interprétation

```
;;; vrai : -> Booleen
;;; faux : -> Booleen
;;; vrai? : Booleen -> bool
;;; faux? : Booleen -> bool
;;; non : Booleen -> Booleen
;;; et : Booleen * Booleen -> Booleen
;;; ou : Booleen * Booleen -> Booleen
```

9.3. Évaluation d'une expression booléenne constante

9.3.1. Spécification

Exactement comme pour les expressions booléennes simples :

```
;;; ebg-exp-const-val : ExprBoolGen/constante/ -> Booleen
;;; ERREUR lorsque «exp» n'est pas une expression constante bien formée
;;; (ebg-exp-const-val exp) rend la valeur de «exp»
```

Exemples :

```
(ebg-exp-const-val '(@et (@ou @v @f @f) (@non @f) @v)) → VRAI
(ebg-exp-const-val '(@et (@ou @v @f @f) (@non @f) @v (@et @v @f))) → FAUX
```

9.3.2. Implantation

Comme d'habitude, le schéma de la défi nition suit la grammaire :

```
(define (ebg-exp-const-val exp)
  (cond ((ebg-atomique? exp)
         ... à faire
         ... à faire
```

```
         ... à faire
         ... à faire
         ... à faire
        ((ebg-negation? exp)
         ... à faire
        ((ebg-conjonction? exp)
         ... à faire
        ((ebg-disjonction? exp)
         ... à faire
        (else (erreur 'ebg-exp-const-val
                      "mal formée ou non constante"))))
```

pour les constantes et les négations, on fait comme pour les expressions simples :

```
(define (ebg-exp-const-val exp)
  (cond ((ebg-atomique? exp)
         (if (ebg-constante? exp)
             (if (ebg-constante-vrai? exp)
                 (vrai)
                 (faux))
             (erreur 'ebg-exp-const-val exp " est une variable")))
        ((ebg-negation? exp)
         (non (ebg-exp-const-val (ebg-neg-sous-exp exp))))
        ((ebg-conjonction? exp)
         ... à faire
        ((ebg-disjonction? exp)
         ... à faire
        (else (erreur 'ebg-exp-const-val
                      "mal formée ou non constante"))))
```

En revanche, les conjonctions et les disjonctions ne peuvent pas être traitées comme dans le cas des expressions simples puisque l'on peut avoir un nombre quelconque de sous-expressions.

Calculer la valeur d'une conjonction paraît compliqué : nous spécifi ons (et nous implanterons) une fonction pour le faire :

```
(define (ebg-exp-const-val exp)
  (cond ((ebg-atomique? exp)
         (if (ebg-constante? exp)
             (if (ebg-constante-vrai? exp)
                 (vrai)
                 (faux))
             (erreur 'ebg-exp-const-val exp " est une variable")))
       ((ebg-negation? exp)
         (non (ebg-exp-const-val (ebg-neg-sous-exp exp))))
        ((ebg-conjonction? exp)
         (ebg-conjonction-const-val (ebg-n-aire-sous-exps exp)))
        ((ebg-disjonction? exp)
         ... à faire
        (else (erreur 'ebg-exp-const-val
                      "mal formée ou non constante"))))
```

```
;;; ebg-conjonction-const-val : LISTE[ExprBoolGen] -> Booleen
;;; ERREUR lorsqu'un des éléments de «exps» n'est pas une expression constante bien formée
;;; (ebg-conjonction-const-val exps) rend la valeur de la conjonction de tous les
;;; éléments de «exps»
```

Noter que la donnée de la fonction `ebg-conjonction-const-val` est une liste d'expressions et, lors de son

application dans `ebg-exp-const-val`, elle est appelée avec la liste des sous-expressions de l'expression donnée.

On agit de même pour les disjonctions :

```
(define (ebg-exp-const-val exp)
  (cond ((ebg-atomique? exp)
         (if (ebg-constante? exp)
             (if (ebg-constante-vrai? exp)
                 (vrai)
                 (faux))
             (erreur 'ebg-exp-const-val exp " est une variable")))
        ((ebg-negation? exp)
          (non (ebg-exp-const-val (ebg-neg-sous-exp exp))))
         ((ebg-conjonction? exp)
          (ebg-conjonction-const-val (ebg-n-aire-sous-exps exp)))
         ((ebg-disjonction? exp)
          (ebg-disjonction-const-val (ebg-n-aire-sous-exps exp)))
         (else (erreur 'ebg-exp-const-val
                       "mal formée ou non constante"))))
```

```
;;; ebg-disjonction-const-val : LISTE[ExprBoolGen] -> Booleen
;;; ERREUR lorsqu'un des éléments de «exps» n'est pas une expression constante bien formée
;;; (ebg-disjonction-const-val exps) rend la valeur de la disjonction de tous les
;;; éléments de «exps»
```

9.3.3. Implantation de `ebg-conjonction-const-val`

Rappelons la spécifi cation :

```
;;; ebg-conjonction-const-val : LISTE[ExprBoolGen] -> Booleen
;;; ERREUR lorsqu'un des éléments de «exps» n'est pas une expression constante bien formée
;;; (ebg-conjonction-const-val exps) rend la valeur de la conjonction de tous les
;;; éléments de «exps»
```

La donnée de cette fonction étant une liste, on suit le schéma sur les listes :

```
(define (ebg-conjonction-const-val exps)
  (if (pair? exps)
      ; valeur lorsque la liste n'est pas vide
      ... à faire
      ... à faire
      ; valeur lorsque la liste est vide
)
```

Comme nous l'avons dit plus haut, la valeur de la conjonction d'une liste d'expressions vide est `VRAI` :

```
(define (ebg-conjonction-const-val exps)
  (if (pair? exps)
      ; valeur lorsque la liste n'est pas vide
      ... à faire
      ... à faire
      (vrai))
)
```

Lorsque la liste n'est pas vide, la valeur de vérité de la conjonction de ses éléments est égale à la conjonction (binaire) de la valeur de vérité de son premier élément et de la valeur de vérité de la conjonction (généralisée) de ses autres éléments. On pourait donc utiliser la fonction `et`, mais, pour des raisons d'effi cacité, on préfère calculer la conjonction binaire à l'aide d'une alternative :

```
(define (ebg-conjonction-const-val exps)
```

```
      (if (pair? exps)
          (if (vrai? (ebg-exp-const-val (car exps)))
              (ebg-conjonction-const-val (cdr exps))
              (faux))
          (vrai)))
```

Remarque : nous pourrions aussi écrire la défi nition suivante :

```
(define (ebg-conjonction-const-val exps)
  (reduce et (vrai) (map ebg-exp-const-val exps)))
```

Nous avons préféré la version donnée ci-dessus car elle est plus effi cace. En effet, dans cette version, dès qu'une sous-expression est fausse, on ne calcule pas la valeur de vérité des autres sous-expressions alors que dans la version avec `map` et `reduce`, on calcule toujours la valeur de vérité de toutes les sous-expressions.

9.3.4. Implantation de `ebg-disjonction-const-val`

Laissée en exercice.

9.4. Simplifi cation d'une expression booléenne avec inconnues

9.4.1. Spécification

Exactement comme pour les expressions simples :

```
;;; ebg-exp-simplifiee : ExprBoolGen -> ExprBoolGen
;;;  ERREUR lorsque «exp» n'est pas une expression bien formée
;;; (ebg-exp-simplifiee expr) rend une expression booléenne simplifiée équivalente à
;;; l'expression booléenne donnée.
```

Exemples :

```
(ebg-exp-simplifiee '(@et @v a b))
→ (@et a b)
(ebg-exp-simplifiee '(@et (@non (@et @v a)) (@ou b (@et @v a))))
→ (@et (@non a) (@ou b a))
```

9.4.2. Implantation

Encore une fois, nous suivons le même schéma :

```
(define (ebg-exp-simplifiee exp)
  (cond
    ((ebg-atomique? exp)
     ... à faire
    ((ebg-negation? exp)
     ... à faire
    ((ebg-conjonction? exp)
     ... à faire
     ... à faire
    ((ebg-disjonction? exp)
     ... à faire
     ... à faire
    (else (erreur 'ebg-exp-simplifiee "expression mal formée"))))
```

Comme pour les expressions simples, la simplifi ée d'une expression atomique est égale à elle-même :

```
(define (ebg-exp-simplifiee exp)
  (cond
    ((ebg-atomique? exp)
     exp)
    ((ebg-negation? exp)
```

```
    ((ebg-conjonction? exp)
     ... à faire
     ... à faire
    ((ebg-disjonction? exp)
     ... à faire
     ... à faire
    (else (erreur 'ebg-exp-simplifiee "expression mal formée"))))
```

les négations se traitent exactement comme dans le cas des expressions simples, aussi nous ne les revoyons pas :

```
(define (ebg-exp-simplifiee exp)
  (cond
    ((ebg-atomique? exp)
     exp)
    ((ebg-negation? exp)
     (ebg-neg-simpl (ebg-exp-simplifiee (ebg-neg-sous-exp exp))))
    ((ebg-conjonction? exp)
     ... à faire
     ... à faire
    ((ebg-disjonction? exp)
     ... à faire
     ... à faire
    (else (erreur 'ebg-exp-simplifiee "expression mal formée"))))
```

Pour les conjonctions (et il en va de même pour les disjonctions) on doit effectuer une opération de simplification sur les simplifiées de toutes les sous-expressions. Pour avoir la liste des simplifiées de toutes les sous-expressions, il suffit de « maper » la fonction `ebg-exp-simplifiee` sur la liste des sous-expressions :

```
(define (ebg-exp-simplifiee exp)
  (cond
    ((ebg-atomique? exp)
     exp)
    ((ebg-negation? exp)
     (ebg-neg-simpl (ebg-exp-simplifiee (ebg-neg-sous-exp exp))))
    ((ebg-conjonction? exp)
     (ebg-conj-simpl (map ebg-exp-simplifiee
                          (ebg-n-aire-sous-exps exp))))
    ((ebg-disjonction? exp)
     (ebg-disj-simpl (map ebg-exp-simplifiee
                          (ebg-n-aire-sous-exps exp))))
    (else (erreur 'ebg-exp-simplifiee "expression mal formée"))))
```

la fonction `ebg-conj-simpl` ayant comme spécification :

```
;;; ebg-conj-simpl : LISTE[ExprBoolGen/simplifiee/] -> ExprBoolGen/simplifiee/
;;; (ebg-conj-simpl exps) rend une expression booléenne simplifiée équivalente à
;;; la conjonction des éléments de la liste «exps»
```

(Le cas de la disjonction est similaire, nous ne le traiterons pas)

9.4.3. Implantation de `ebg-conj-simpl`

Comment peut-on simplifier une conjonction ?

- si une des sous-expressions est fausse, l'expression est fausse,
- on peut supprimer toutes les sous-expressions vraies,
- une conjonction sans sous-expressions est vraie,
- une conjonction ayant une seule sous-expression est équivalente à cette sous-expression.

D'où l'implantation de `ebg-conj-simpl` :

```
(define (ebg-conj-simpl exps)
  (if (member (ebg-constante-faux) exps)
      (ebg-constante-faux)
      (let ((exps-simp (ebg-liste-sans (ebg-constante-vrai) exps)))
        (if (pair? exps-simp)
            (if (pair? (cdr exps-simp))
                (ebg-conjonction exps-simp)
                (car exps-simp))
            (ebg-constante-vrai)))))
```

la fonction `ebg-liste-sans` ayant comme spécifi cation :

```
;;; ebg-liste-sans : alpha * LISTE[alpha] -> LISTE[alpha]
;;; (ebg-liste-sans el ll) rend la liste des éléments de «ll» qui ne sont pas égaux à «el»
```

L'implantation de cette fonction, simple exercice sur les listes, est laissée en exercice (on pourra utiliser la fonction `filtre`).