λProlog - Intelligence Artificielle

Yves Bekkers

24 May 2011

Avertissement. Ce cours s'adresse à des étudiants qui ont déjà une bonne connaissance de la programmation déclarative à travers au moins un des deux langages suivants : un langage fonctionnel tel que CAML ou un langage relationnel tel que PROLOG. Il n'y a pas besoin de connaître les deux.

1 Introduction

Nous considérons qu'il existe deux grandes familles de langages informatique :

- Les langages impératifs qui miment le fonctionnement des unités centrales des ordinateurs et de leur mémoire.
- Les langages déclaratifs qui s'inspirent de la syntaxe et de la sémantique de théories mathématiques. Bien connues avant l'arrivée de l'informatique, ces théories mathématiques ont été remisent au goût du jour pour les besoins de l'informatique.

1.1 Langages impératifs

Un *programme impératif* est une *séquence d'ordres*. Voici l'ancêtre des langages impératifs, l'assembleur :

```
LD,R1 mem(1234)   // lire une valeur en mémoire,
ADD,R1 1          // effectuer une opération arithmétique,
STD,R1 mem(1234)  // ranger la nouvelle valeur en mémoire,
GOTO mem(4567)    // opérer une rupture de contrôle de séquence,
```

Le modèle de calcul est la *Machine de Turing*. Le principe opératoire d'un langage impératif est «*Exécuter la séquence d'ordres pour faire changer l'état de la mémoire de l'ordinateur*». Les langages orientés objet se classent dans cette catégorie.

Le principe fondamental des langages impératifs se résume dans l'instruction :

```
x := x+1.
```

Du point de vue des mathématiques cette égalité est une aberration. Il s'agit d'une «*pseudo-égalité*» inventée par les informaticiens pour effectuer de la réutilisation de mémoire. Il y a l'état de la mémoire avant et l'état après l'exécution de l'instruction. Il s'agit d'un ordre de gestion de mémoire explicite.

Les mots clés des langages impératifs sont «*séquences, état mémoire*».

TABLEAU 1. Exemples de langages impératifs

ASSEMBLEUR
FORTRAN
PASCAL
C, C++
JAVA
JAVASCRIPT

1.2 Langages déclaratifs

Là où les langages impératifs miment le fonctionnement impératif de l'unité centrale d'un ordinateur, un langage déclaratif mime une théorie mathématique, tant pour sa syntaxe que pour sa sémantique.

Les langages déclaratifs possèdent tous des traits impératifs plus ou moins prononcés, mais ils conservent cependant toujours la particularité d'avoir été conçus au départ avec une théorie mathématique comme modèle. La force des langages déclaratifs sur leurs ancêtres impératifs est que les premiers permettent généralement d'attaquer une même programmation sous deux angles de vision très différents, mais complémentaires, à savoir :

1. la vision déclarative qui donne une approche «spécification» de la programmation ;
2. la vision impérative qui s'attache plus à la mise en oeuvre effective et aux problèmes de performance.

Il existe deux grandes familles de langages déclaratifs, ceux basés sur la notion de fonction et ceux basés sur la notion de relation, en voici des exemples :

TABLEAU 2. Deux familles de langages déclaratifs

	Fonctionnels	**Relationnels**
Exemples	SCHEME, ML, OCAML	PROLOG, CLP, λPROLOG
Caractérisation	Fonction, Application	Relation, Unification
Modèle	λCalcul	Logique des prédicats du 1^{ier} ordre

Un *programme déclaratif* est un *problème* posé dans la syntaxe d'un langage mathématique.

Le principe opératoire des langages déclaratifs est «*Trouver la solution au problème posé en utilisant la sémantique du langage utilisé*».

Un des mots clés des langages déclaratifs est *stratégie* :

- Pour les langages fonctionnels les stratégies définissent quand et comment sont évaluées les expressions servant de paramètres effectifs aux fonctions ;

- Pour les langages relationnels (langages de programmation logiques) nous verrons que les stratégies portent d'un coté, sur le choix des clauses ou règles du programme (choix OU), de l'autre sur le choix des buts dans les questions (choix ET).

1.2.1 Langage fonctionnels

Un programme en langage fonctionnel est composé d'un ensemble de fonctions qui définissent un *contexte*, et d'une expression à évaluer dans ce contexte. L'expression est le problème à résoudre. Le résultat de l'évaluation de l'expression (sa réduction) est la solution au problème posé. Voici quelques exemples

TABLEAU 3. Exemples de programmes fonctionnels

Programme	Expression à évaluer	Solution
L'arithmétique sur les entiers	`1 + 3`	*int* `: 4`
L'arithmétique sur les entiers	`2 > 1`	*boolean* `: true`
L'arithmétique sur les entiers, plus la conditionnelle	`if 2>1 then "oui" else "non" fsi`	*string* `: "oui"`

1.2.2 Langages relationnels

Un programme relationnel est composé d'un ensemble d'axiomes qui définissent une théorie, et d'un problème qui est un théorème à démontrer dans la théorie énoncée par le programme.

Une logique propose une syntaxe pour représenter le monde réel, nous allons utiliser un sous ensemble de la syntaxe du calcul des prédicats, voici des exemples :

TABLEAU 4.

Fait du monde réel	**En calcul des prédicats**
«Félix est un chat»	chat(felix)
«tout chat ronronne»	$(\forall x)(\text{ronronne}(x) \Leftarrow \text{chat}(x))$
«quelqu'un ronronne»	$(\exists x)\text{ronronne}(x)$
«Félix est le père de Noiraud»	pere(felix,noireau)
«pour être une mère il suffit qu'il existe quelqu'un dont on soit la mère»	$(\forall x)(\text{est} - \text{mere}(x) \Leftarrow (\exists z)\text{mere}(x,z))$

Conjointement avec la syntaxe qu'elle propose une logique offre des procédés de démonstration qui miment les raisonnements du monde réel. Le calcul des prédicats dans toute sa généralité n'a pas de procédé automatique effectif de démonstration permettant de décider si une formule est un théorème d'une théorie. Effectif voulant dire qui se termine à tous les coups.

Les langages de programmation logiques utilisent une syntaxe qui est un sous ensemble du calcul des prédicats. Les limites portent à la fois sur la syntaxe des axiomes ainsi celles des théorèmes à démontrer. Sur ce sous ensemble bien

défini, les langages de programmation logiques mettent en oeuvre un procédé qui n'est pas un démonstrateur complet mais qui a la propriété suivante :

> *A partir d'un programme P représentant une théorie T et à partir d'une question Q, si le procédé s'arrête avec un succès, on peut dire que Q représente un théorème de T.*
>
> *Si le procédé boucle ou s'il s'arrête avec échec on ne peut rien dire sur Q.*

Dans la prochaine section nous introduisons brièvement PROLOG et λPROLOG. Dans les sections à venir nous reconstruirons progressivement la syntaxe abstraite de PROLOG et nous en donnons une syntaxe concrète, celle de λPROLOG.

2 PROLOG et λPROLOG, notion de prédicat

Les langages de programmation logique permettent de représenter (modéliser) le monde réel. Leur syntaxe est définie en deux niveaux :

1. Les *termes* qui servent à représenter les objets du monde réel, par exemple *Pierre* une personne, *Félix* un chat.
2. Les *formules* qui permettent de représenter des énoncés du monde réel. Par exemple «*Pierre est une personne*», «*Félix est une chat*».

PROLOG et λPROLOG sont deux langages de la famille des langages de programmation logique aussi appelés langages relationnels. Leur modèle de calcul est la logique des prédicats du premier ordre. Cependant ces deux langages diffèrent tant au niveau des formules qu'au niveau des termes qu'ils manipulent. Voici un tableau les comparant sur ces deux plants

TABLEAU 5. Prolog et λProlog

	Prolog	**λProlog**
Formules logiques	Clauses de Horn	Formules Héréditaires de Harrops
Termes	Termes du premier ordre	Termes simplement typés du λcalcul
Unification	Termes du premier ordre	Résolution de contraintes sur les termes simplement typés du λcalcul

Syntaxe abstraite Du point de syntaxe abstraite λPROLOG est une double extension de PROLOG tant au niveau des termes qu'au niveau de formules.

Syntaxe concrète Du point de vue Syntaxe concrète PROLOG est une simple concrétisation de la syntaxe de la logique. Nous verrons que λPROLOG utilise quand à lui un syntaxe fonctionnelle currifiée dans la lignée des langages

fonctionnels tels que OCAML. Voici une comparaison de trois syntaxes concrètes pour représenté un énoncé atomique (sans connecteur logique) :.

TABLEAU 6. Représenter un énoncé

Enoncée	*Pierre est le père de Jacques*
Logique	`pere(pierre,jacques)`
PROLOG	`pere(pierre,jacques)`
λPROLOG	`pere pierre jacquesprolog`

2.1 Programmer en logique

La programmation logique repose sur la définition de *prédicats* ou *relations* portant sur des objets syntaxiques appelés des termes. Les *termes* sont des représentations symboliques des objets de ce monde réel. Les formules permettent de représenter les relations.

2.1.1 La théorie «*rose des vents*»

Considérons l'énoncé `E3` ci-dessus. Nous allons modéliser le monde de cet énoncé; Cet énoncé parle d'une relation que l'on peut spécifier par le tableau suivant

TABLEAU 7. Relation «la rose des vents»

Point cardinal	**Adjectif latin**
nord	boréal
sud	austral
est	oriental
ouest	occidental

La représentation de relations par des tables est empruntée aux bases de données relationnelle. PROLOG, λPROLOG et tous les autres langages de programmation logique nous permettent de modéliser des relations par des formules logiques simples.

En logique nous modélisons la théorie «*rose des vents*» par quatre *assertions* ou *faits*

TABLEAU 8. La rose des vents en logique

	Assertion logique	**Enoncé**
A1	*roseDesVents(nord,boreal)*	*boreal est l'adjectif correspondant à la direction nord*
A2	*roseDesVents(sud,austral)*	*austral est l'adjectif correspondant à la direction sud*
A3	*roseDesVents(est,oriental)*	*oriental est l'adjectif correspondant à la direction est*
A4	*roseDesVents(ouest,occidental)*	*occidental est l'adjectif correspondant à la direction ouest*

2.1.2 Prédicat, arité d'un prédicat

La conjonction des quatre assertions ci dessus définit ce que l'on appelle un *prédicat*. Un prédicat est un groupe d'assertions qui portent toutes le même nom,

ici *roseDesVents*, et le même nombre d'arguments. Le prédicat *roseDesVents* est *binaire* car il a deux arguments. On dit qu'il a une *arité* de 2. Le prédicat *estUnHomme* dans l'assertion *estUnHomme(socrate)* est unaire, il est d'arité 1.

Le prédicat est à la logique ce que la relation est aux bases de données relationnelles. Une relation porte un nom et un nombre d'attributs typés. un prédicat possède un nom et un nombre d'arguments typés eux aussi.

2.2 Représenter et typer les entités du monde réel

Pour mieux appréhender la complexité de monde réel et ses entités l'homme a toujours essayé de classer les entités et de les nommer, c'est ce que l'on appel depuis le XVIII siècle la *taxonomies* ou *taxinomie*. Dans l'exemple de la rose des vents, nous avons deux types d'entités, deux *taxons*, à savoir les points cardinaux d'un coté et leurs adjectifs correspondants de l'autre. Grâce à son système de type robuste λPROLOG va nous permettre de rendre compte dans nos programmes de cette taxinomie et ainsi de rendre nos programme plus robustes.

En logique les huit entités manipulées peuvent être représentées directement par le mot français qui les représentent. Ainsi cependant ils perdent leur taxinomie. En programmation les choix de représentation sont nombreux. Deux méthodes principales de représentation s'imposent en programmation déclarative :

- Les *chaînes*, que l'on retrouve comme type primitif dans tous les langages de programmation. Par exemple le point cardinal *nord* est représenté par la chaîne `"nord"`, l'adjectif boréal par la chaîne `"boreal"`. Avec cette méthode les entités perdent leur taxonomie.
- Les identificateurs de *termes simples* aussi appelés *atomes*. Par exemple le point cardinal *nord* est représenté par l'identificateur de terme `nord` ou encore `Nord`.

Nous préférons la seconde solution car en utilisant un langage typé on pourra rendre compte de la taxonomie des entités modélisées. C'est ce que nous voyons maintenant.

Les langages déclaratifs se classent en deux familles, ceux qui sont typés et ceux qui ne le sont pas, PROLOG se range dans la famille des langages non typés et λPROLOG et un langage typé :

TABLEAU 9. Langages déclaratifs

	Programmation fonctionnelle	**Programmation logique**
non typés	SCHEME, LISP	PROLOG, CLP
typés	ML, OCAML	λPROLOG

2.2.1 OCAML

Voici les déclarations de deux types en OCAML qui rendent compte de la taxonomie du monde de la rose des vents :

```
type cardinal = Nord | Sud | Est| ouest
type adjectif = Boreal | Austral | Oriental | Occidental
```

Il s'agit de deux déclarations de type énuméré. En OCAML les constantes commencent par une majuscule.

2.2.2 PROLOG

En PROLOG il est impossible de typé les termes que nous manipulons. Ainsi utiliser des chaînes ou des termes ne fait pas de différence.

2.2.3 λPROLOG

Voici les déclarations de types énumérés en λPROLOG :

```
kind cardinal type.
type (nord, sud, est, ouest) cardinal.

kind adjectif type.
type (boreal, austral, oriental, occidental) adjectif.
```

En λPROLOG la déclaration se fait en deux temps.

1. Tout d'abord, par la directive `kind`, vient la déclaration des taxons (au sens taxinomie) `cardinal` et `adjectif` (vous pouvez aussi les appeler *classes d'objets* ou *types*). In fine chaque taxon ou type dénotera un ensemble de valeurs de même type. Pour λPROLOG `cardinal` et `adjectif` sont ce qu'on appel des *types simple*s. Nous verrons qu'il est possible de construire en λPROLOG comme en OCAML des taxonomies plus complexes.
2. Puis vient la déclaration des valeurs et de leur type par la directive `type`. En λPROLOG les valeurs commencent par des minuscules.

2.3 Modèles de calcul

Les calculs en OCAML sont exprimés par des fonctions. En λPROLOG nous allons utiliser des relations ou *prédicats.*

2.3.1 Fonction `dir_to_adjectif` en OCAML

Voici la définition complète en OCAML de la fonction `(dir_to_adjectif d)` qui rends rend l'adjectif correspondant à une direction donnée `d` :

```
type cardinal = Nord | Sud | Est| ouest
type adjectif = Boreal | Austral | Oriental | Occidental

type dir_to_adjectif cardinal -> adjectif.

let dir_to_adjectif d = match d with
   | Nord -> Boreal
   | Sud -> Austral
   | Est -> Oriental
   | Ouest -> Occidental
```

Pour définir cette fonction à un argument OCAML utilise une variable `d` (concrètement un identificateur commençant par une minuscule), qui représente le paramètre de la fonction. Le paramètre est *filtré* par l'expression `match d with ...`.

Notions de programme et d'exécution Un programme fonctionnel définit un ensemble de fonctions qui constituent un contexte d'évaluation. Une exécution est la réduction (l'évaluation) d'une expression dans le contexte défini par le programme.

Exemple : dans le contexte défini par la fonction `dir_to_adjectif`, ci-dessus, une expression telle que `(dir_to_adjectif Nord)` se réduit en la valeur `Boreal`.

Ce principe vous est déjà connu.

Remarquez que pour obtenir une direction à partir d'un adjectif une seconde fonction OCAML doit être écrite.

2.3.2 Prédicat `roseDesVents` en λPROLOG

La fonction unaire `dir_to_adjectif` de OCAML devient un prédicat binaire `roseDesVents` en λPROLOG. Le programme complet avec ses déclarations de type se présente comme suit :

```
kind cardinal type.
type (nord, sud, est, ouest) cardinal.

kind adjectif type.
type (boreal, austral, oriental, occidental) adjectif.

type roseDesVents cardinal -> adjectif -> o.

roseDesVents nord boreal.
roseDesVents sud austral.
roseDesVents est oriental.
roseDesVents ouest occidental.
```

Remarquez la déclaration du symbole `roseDesVents`. Il s'agit de la déclaration d'un prédicat à deux arguments. Le nombre d'arguments est donné par le nombre de flèches dans la déclaration. A l'image des déclarations de fonctions en OCAML, chaque flèche d'une déclaration de prédicat introduit un nouvel argument. Le premier argument du prédicat `roseDesVents` est de type `cardinal`, le second est de type `adjectif`.

Une expression telle que `(roseDesVents nord boreal)` est une formule logique atomique (elle ne contient pas de connecteur logique). Le taxon (type) des formules logiques en λPROLOG est prédéfini il s'agit du type `o`, le type pour les formules logiques.

Du fait de l'utilisation de la notation currifiée les expressions :

```
(roseDesVents nord boreal)
((roseDesVents nord) boreal)
```

sont équivalentes en λPROLOG. On dit que *l'application* est associative à gauche. La flèche quand à elle, dans les déclarations de type, est associative à droite de sorte que les deux déclarations suivantes sont équivalentes :

```
type roseDesVents cardinal -> adjectif -> o.
type roseDesVents cardinal -> (adjectif -> o).
```

Nous verrons que l'expression `(roseDesVents nord)` de type `adjectif -> o` est un prédicat unaire en λPROLOG. Il est manipulable au même titre que des expressions telles que `nord` ou `(roseDesVents nord boreal)`. Nous verrons ainsi que la séquence de code qui suit est parfaitement correcte en λPROLOG :

```
X = (roseDesVents nord), (X boreal).
```

Dans cette séquence la variable `X` reçoit comme valeur un prédicat unaire de type `(adjectif -> o)`. Ce prédicat est ensuite appliqué à un argument du type `adjectif`.

Voici des expressions manipulables par λPROLOG et correctement typées. Nous donnons leur type correspondant :

TABLEAU 10. Exemples d'expressions et leur type

Expression	Type de l'expression
`boreal`	`adjectif`
`nord`	`cardinal`
`roseDesVents`	`cardinal -> adjectif -> o`
`(roseDesVents nord)`	`adjectif -> o`
`(roseDesVents nord boreal)`	`o`

2.3.3 Formules atomiques, fait ou but ?

En programmation logique on appel une formule logique atomique un *fait* ou un *but* selon le contexte. Les formules atomiques de PROLOG et λPROLOG suivent une syntaxe particulière appelée *syntaxe des termes*. Nous en donnons la définition formelle en "Syntaxe des termes du premier ordre", page 17.

Un fait ou un but tel que `(roseDesVents nord boreal)` peut représenter les deux énoncés totalement opposés suivants :.

TABLEAU 11. Fait ou négation d'un fait

	λPROLOG	Enoncé
Fait	`(roseDesVents nord boreal)`	*boreal est l'adjectif correspondant à la direction nord*
But	`(roseDesVents nord boreal)`	*Est-ce que boreal est l'adjectif correspondant à la direction nord ?*

Dans les programmes logiques cette ambiguïté d'interprétation n'est jamais un problème. Elle est toujours levée par la position du fait dans le programme. Nous verrons cela plus loin.

Dans le présent texte nous représenterons les questions en les préfixant du symbole `:-` qui mime le symbole «*turnstile*» («**`\vdash`**» en LATEX, «**`right tack`**» en UNICODE dont le code hexadécimal est `#22A2`). En programmation logique, lorsqu'une formule logique atomique représente une assertion. on l'appel un *fait*, lorsqu'il s'agit d'une question ou d'un théorème a prouver, on l'appel un *but*.

2.3.4 Représenter une théorie par un programme

Du point de vue logique une théorie est un ensemble d'axiomes énoncés sous forme de formules logiques. Cet ensemble peut être vu comme la conjonction des formules logiques qu'il contient.

La théorie de la «*rose des vents*» ci-dessus est constituée de l'ensemble d'axiomes suivant :

TABLEAU 12. Axiomes de la théorie «*rose des vents*»

	Axiomes
A_1	*roseDesVents(nord,boreal)*
A_2	*roseDesVents(sud,austral)*
A_3	*roseDesVents(est,oriental)*
A_4	*roseDesVents(ouest,occidental)*

Si `T` est la théorie «*rose des vents*», notera $T = \{A_1,A_2,A_3,A_4\}$.

En λPROLOG la théorie de la «r*ose des vents*» est représentée par le programme $P = \{F_1,F_2,F_3,F_4\}$ constitué des quatre faits suivants :

```
roseDesVents nord boreal.      /* F1 */
roseDesVents sud austral.      /* F2 */
roseDesVents est oriental.     /* F3 */
roseDesVents ouest occidental. /* F4 */
```

Ces quatre faits définissent un unique prédicat, le prédicat `roseDesVents`. On peut le voir comme une relation. Comme une relation, un prédicat porte un nom, ici «`roseDesVents`». Généralement un programme logique (programme relationnel) définit un ensemble de prédicats.

Nous voyons maintenant comment la logique d'un coté et l'interpréteur λPROLOG répondent une question simple énoncée comme suit :

«Est-ce que boréal est l'adjectif qui correspond à la direction nord ?»

TABLEAU 13. Questions en logiques et en PROLOG

Logique	**Prolog**
Montrez que *roseDesVents(nord,boreal)* est un théorème de la théorie «*rose des vents*».	A partir des faits du programme P démontrez le but suivant : `:-` (roseDesVents `nord boreal`).

3 Début de reconstruction du langage PROLOG

Dans cette partie nous commençons la reconstruction à partir de rien lu sous ensemble PROLOG du langage λPROLOG. Nous utilisons la syntaxe concrète currifiée de λPROLOG mais nous laisserons temporairement de coté les nouveaux connecteurs logiques introduits par λPROLOG.

Pendant cette reconstruction nous mènerons en parallèle deux moyens de comprendre le fonctionnement d'un interpréteur PROLOG :

- Par la logique, de nombreuses méthodes de démonstrations syntaxiques sont à notre disposition, nous utiliserons ici une logique appelée la *déduction naturelle* car elle est très intuitive et comporte un petit nombre de règles de déduction simples à comprendre.
- Par le fonctionnement opératoire. Nous décrirons le fonctionnement de PROLOG par une technique d'effacement et de remplacement de but.

3.1 Première étape de reconstruction - où on appelle un chat un chat

Dans cette section nous considérons exclusivement des formules logiques atomiques, c'est à dire sans les connecteurs logiques $\wedge$, $\vee$ et $\Rightarrow$ sans les quantifications.

Voici un tableau de quelques énoncés sans quantification du monde réel :

TABLEAU 14. Quelques énoncés du monde réel

E1	*Félix est un chat*
E2	*Boréal est l'adjectif qualificatif correspondant à la direction géographique nord.*
E3	*Les éléphants sont plus gros que les chevaux.*

Voici la retranscription de ces énoncés en calcul des prédicats :

TABLEAU 15. Quelques formules atomiques

E1	*estUnChat(felix)*
E2	*roseDesVents(nord,boreal)*
E3	*plusGros(elephants,chevaux)*

3.1.1 Point de vue logique - règle d'identité

La déduction naturelle dispose de règles qui permettent de déduire des théorèmes qui sont conséquence logique des axiomes de la théorie. La première règle que nous voyons est appelée *règle d'identité* ou *axiome.* Elle s'énonce ainsi :

«*A partir d'un axiome A on peut déduire le théorème A.*»

Cette règle s'écrit visuellement comme suit :

$$\frac{A}{A}I \qquad \text{ou encore} \qquad \frac{}{A}A$$

FIGURE 1. Règle d'identité ou axiome

En déduction naturelle cette règle s'applique à toute formule logique A. Nous sous-employons donc ici cette règle car nous ne parlons dans cette première étape de reconstruction que de formules logiques atomiques.

Le haut d'une la règle telle que ci-dessus s'appel les *prémisses* le bas la *conclusion*. Les prémisses c'est ce qui a déjà été prouvé ou qui est un axiome et la conclusion c'est le nouveau théorème que l'on prouve par la règle. Grâce à la règle d'identité, à partir d'une théorie composée de la conjonction de n axiomes $\{A_1,A_2,\ldots,A_n\}$ nous pouvons déduire n'importe lequel des théorèmes A_1, A_2, ..., A_n.

Ainsi nous déduisons que *roseDesVents(sud,austral)* est un théorème dans la théorie «*rose des vents*» de la manière suivante :

$$\frac{\text{roseDesVents(sud,austral)}}{\text{roseDesVents(sud,austral)}}I \qquad \text{ou encore} \qquad \frac{}{\text{roseDesVents(sud,austral)}}A$$

Remarque sur la négation Par contre la logique et notre règle de déduction ne nous disent rien sur la formule *roseDesVents(sud,boreal)*. Nous sommes incapable de décider s'il s'agit ou non d'un théorème de la théorie «*rose des vents*». Nous ne pouvons pas non plus affirmer que $\neg$*(roseDesVents(sud, boreal))* est un théorème de la théorie «*rose des vents*». Voir Section 3.1.3, page 13 pour une discutions sur ce sujet brûlant de la négation en programmation logique.

3.1.2 Point de vue opératoire - règle d'effacement

Pour mimer la règle de la déduction naturelle que nous venons de présenter λPROLOG explore les faits du programme qui lui sont donnés (représentant la théorie considérée) et confronte ces faits avec le but (représentant le théorème à prouver). Cette confrontation consiste vérifier que le but est égal à un fait du programme. Si c'est le cas l'interpréteur est alors autorisé à «*effacer*» le but. Le but effacé signifie que l'exécution se termine avec *succès*, sinon si le but ne s'efface pas on dit que l'exécution est un échec.

La règle d'effacement est la suivante :

Règle d'effacement :

«On dit qu'un but B atomique s'efface avec succès dans un programme P s'il existe un fait atomique F appartenant au programme P tel que F = B *»*

Ce qui se représente par le schéma suivant P étant un programme composé de formules atomiques et F et B des formules atomiques :

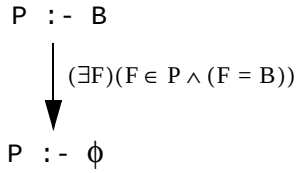


FIGURE 2. Règle d'effacement d'un but par identité

Voici l'effacement du but `:-(roseDesVents sud austral)` dans le programme définissant le prédicat `roseDesVents` ci-dessus

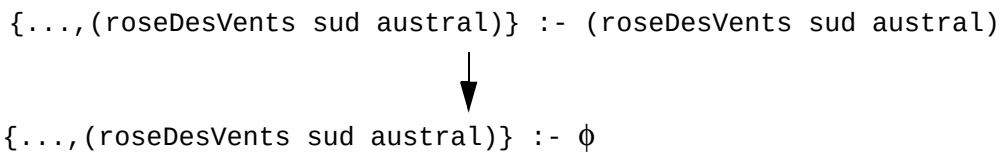


FIGURE 3. Effacement avec succès

L'exécution d'un programme λPROLOG se termine donc de deux manières :

- avec un succès si on arrive à effacer le but
- par un échec si on n'arrive pas à effacer le but.

Par exemple, l'effacement du but `:-(roseDesVents sud boreal)` se termine par un échec dans le contexte du programme `P` ci-dessus car aucun fait de `P` n'est égal à ce but.

Pour arriver à l'échec, l'interpréteur λPROLOG doit explorer tous les faits du programme et vérifier qu'aucun fait n'est égal au but. Il doit donc parcourir tout le programme. Voici des exemples de résultat d'effacement de but dans le contexte du programme *«rose des vents»* :

TABLEAU 16. Succès ou échec de l'effacement de buts dans le programme *«rose des vents»*

But	**Résultat**
`:- rose_des_vents sud austral.`	succès
`:- rose_des_vents sud boreal.`	échec

Remarque sur les types : En λPROLOG contrairement à PROLOG il est impossible de demander la démonstration des buts suivants :

```
:- rose_des_vents sud bleu.
:- rose_des_vents sud sud.
```

Ces formules provoquent des erreurs de syntaxe dès la compilation, les valeurs `bleu` et `sud` n'ayant pas été définies comme des valeurs de type `adjectif`.

3.1.3 PROLOG et la négation

Comme pour la logique, lorsque l'interpréteur PROLOG échoue dans l'effacement d'un but `B` nous n'avons aucun droit d'affirmer que le but représente une formule fausse dans la théorie représentée, c'est à dire que $\neg B$ est un

théorème de la théorie. Nous pouvons juste dire que nous ne savons rien sur `B`. Voici un exemple :

```
kind personne type.

type (pierre, marie, jean) personne.

type grand personne -> o.

grand pierre.
grand marie.
```

Dans ce programme trois personnes sont connues, deux seulement sont définie comme grandes. On ne dit rien sur la taille de Jean la troisième personne. On n'a aucun droit de dire que $\neg$grand(jean) est vrai même si la question `:- grand jean.` se termine par un échec.

Par contre on peut faire l'*hypothèse du monde clos* suivante :

«Tout ce qui n'est pas dit dans le programme est faux.»

Ce qui est facile à implémenter en disant que si un but `B` ne peut s'effacer alors, sous hypothèse du monde clos, $\neg$B est vrai.

Faire l'hypothèse du monde clos sur le programme ci-dessus revient à affirmer «*seuls pierre et marie sont grands*». On est alors en droit de dire que $\neg$grand(jean) est vrai.

A propos de la suite Avant de continuer notre reconstruction progressive de PROLOG nous devons faire une extension importante du langage que nous sommes en train de reconstruire. Nous allons parler des variables et de l'opération d'unification centrale à tout interpréteur PROLOG. Nous nous limiterons dans un premier temps à la syntaxe des termes du premier ordre manipulés par PROLOG, laissant ainsi de coté les λtermes simplement typés manipulés par λPROLOG.

4 Les termes du premier ordre et leur unification

4.1 Des variables dans les questions.

Ici nous enrichissons notre langage avec des buts quantifiés existentiellement.

4.1.1 La théorie «*Socrate est un homme*»

La théorie suivante spécifie que «s*ocrate est un homme*». Elle est composée d'un seul axiome :

estUnHomme(socrate)

La déduction naturelle nous permet de démontrer que le théorème suivant $(\exists x)\text{estUnHomme}(x)$ est un théorème de la théorie «*socrate est un homme*». Il signifie «*il existe un homme*» dans la théorie «*socrate est un homme*».

La déduction naturelle nous offre une règle appelée *règle de généralisation* ou règle *d'introduction de la quantification* ∃. Cette nouvelle règle permet de déduire un théorème existentiel tel que ci dessus à partir d'un axiome qui lui ressemble. Cette règle est la suivante :

$$\frac{\text{estUnHomme(socrate)}}{(\exists x)\text{estUnHomme(x)}}\,GEN$$

FIGURE 4. Règle de généralisation

Si non savons que SOCRATE est un homme dans une théorie, cette règle nous permet de déduire qu'il existe un homme dans cette théorie.

4.1.2 Le programme «*socrate est un homme*»

Pour représenter la théorie «s*ocrate est un homme*» en λPROLOG nous définissons d'abord le taxon `humain` puis la constante `socrate` de type `humain` puis enfin le prédicat `estUnHomme` :

```
kind humain type.
type socrate humain.

type estUnHomme humain -> o.
estUnHomme socrate.
```

En λPROLOG une question existentielle tel que $(\exists x)\text{estUnHomme}(x)$ s'exprime simplement comme suit :

```
:- estUnHomme X.
```

Ici `estUnHomme` est le symbole de prédicat et `X` est une *variable logique*. Les symboles de variable en programmation logique commencent par une majuscule, ce qui les différencient des symboles de prédicat et des autres valeurs qui commencent toujours par une minuscule.

Remarquez que le symbole de quantification ∃ est omis en λPROLOG. Dans un but les variables sont supposées être quantifiée existentiellement implicitement en début de formule.

4.2 Point de vue opératoire, les termes, l'unification

La technique d'effacement de but présentée en "Point de vue opératoire - règle d'effacement", page 12, suppose la mise en œuvre d'une égalité sur les formules atomiques (les formules sans les connecteurs logiques ∀,∃,¬,∧,∨,⇐). Pour l'instant nos formules atomiques ne comportent pas de variables. Décider de l'égalité de deux formules atomiques sans variable ne pose pas de problème.

Un simple parcours en parallèle des deux formules nous permet de vérifier leur égalité. Dans ce qui suit nous introduisons les variables.

4.2.1 Introduction : comparer des termes avec variables

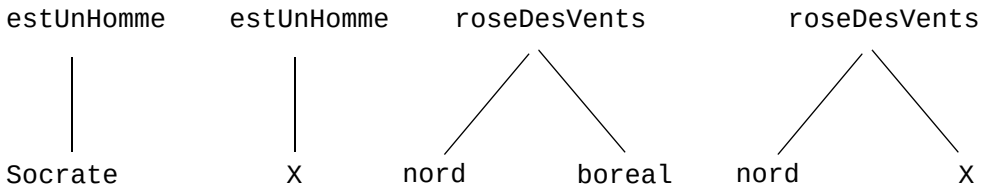
Il nous faut maintenant être capable de comparer deux formules comportant éventuellement des variables, comme par exemple les formules :

`(estUnHomme socrate)` et `(estUnHomme X)`

ou encore les formules :

`(roseDesVents nord boreal)` et `(roseDesVents nord X)`

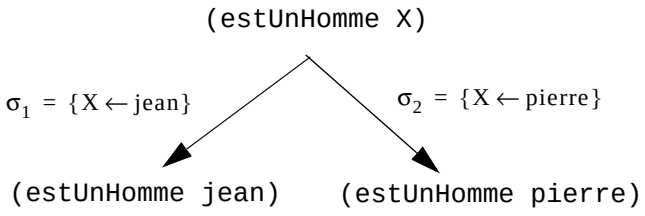
Voici une représentation en arbre de ces quatre formules :



En programmation logique ces formules atomiques sont appelées des *termes*.

Un terme sans variable, appelé *terme clos*, représente une entité, une propriété ou une relation particulière du monde réel.

Un terme avec variable, appelé *terme général*, représente quand à lui une famille d'entités du monde réel ayant des caractéristiques communes. La variable est appelée une *inconnue*. On peut passer d'un terme général à un terme moins général, voir un terme clos, par substitution de variable, voici un exemple :



Pour passer du terme `(estUnHomme X)` au terme `(estUnHomme jean)` on dit que l'on *applique la substitution* σ_2. Ce l'on note :

`(estUnHomme pierre)` = σ_2`(estUnHomme X)`

Voir "Substitution", page 18, pour une définition formelle de la substitution et de l'application d'une substitution à un terme.

L'unification est l'opération qui permet de comparer les termes entre eux même s'ils possèdent des variables. L'unification est au centre de la programmation logique. Pour la définir précisément nous devons d'abord définir

précisément de ce que nous appelons terme. C'est ce que nous faisons maintenant.

4.2.2 Syntaxe des termes du premier ordre

En PROLOG les termes respectent la syntaxe dite des *termes du premier ordre* données ci-dessous. Nous verrons qu'en λPROLOG les termes suivent une syntaxe plus fine et plus souple. Pour l'instant voyons la syntaxe des termes du premier ordre de PROLOG qui est la suivante :

Les termes du premier ordre sont définis comme suit :

Soit U un ensemble de variables (ex : `X, Y, U1`*, ...)*

Soit C0 un ensemble de valeurs ou constructeurs d'arité 0. On les appelle aussi les atomes (ex : `jeanne, felix, ...`*)*

Soit Cn des ensembles de constructeurs d'arité $n \in N \wedge n > 0$ *(ex :* `pere/2, femme/1`*).*

Les formules atomiques A que l'on appelle en logique termes de premier ordre *sont définies récursivement par la grammaire suivante :*

$$A = U | C_0 | C_n(\overbrace{A, \ldots, A}^{n})$$

L'arité n d'un constructeur C_n est le nombre d'arguments que peut prendre le constructeur, par exemple le constructeur `pere` prend deux arguments, de sorte que le terme `pere(jean, jeanne)` est correctement construit. Nous le noterons parfois dans notre discours `pere/2` pour signifier qu'il s'agit d'un constructeur d'arité 2. Dans cette même notation un atome tel que `felix` devient `felix/0`.

Ainsi définis, les termes sont des arbres dont les nœuds sont décorées par les constructeurs et les feuilles sont soit des atomes soit des variables. A chaque noeud, l'arité du constructeur définit le nombre de fils possédés par ce noeud. Voici deux exemples de termes :

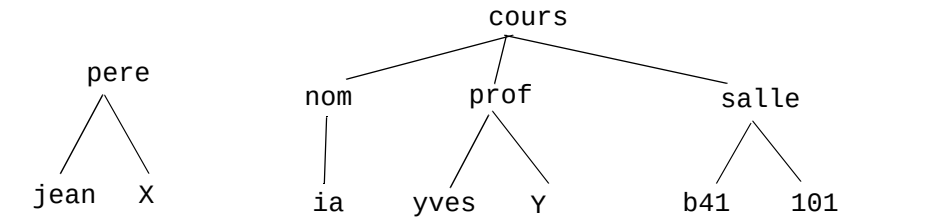


FIGURE 5. Termes construits

Le premier terme est construit à partir du constructeur {`pere/2`} de la variable {`X`} et des atomes {`jean, jeanne`}. Le second est construit à partir des constructeurs {`cours/3, nom/1, prof/2, salle/2`} de la variable {`Y`} et des atomes {`ia, yves, bekkers, b41, 101`}.

On appelle ces termes les termes du premier ordre car les nœuds ne peuvent pas être décorés par des variables, i.e. on ne peut pas écrire `(X jean marie)`.

4.2.3 Substitution

Considérons *A* l'ensemble des termes du premier ordre définis en "Syntaxe des termes du premier ordre", page 17, *U* l'ensemble des variables, une *substitution* est un ensemble de couples $\{x_1 \leftarrow t_1, x_2 \leftarrow t_2, \ldots, x_n \leftarrow t_n\}$ tels que :

$$\forall i(x_i \in U \wedge t_i \in A) \ \wedge \ \forall i \forall j(i \neq j \Rightarrow x_i \neq x_j) \ \wedge \ \forall i \forall j(x_i \notin t_j)$$

La seconde condition interdit deux occurrences de la même variable. La dernière condition est appelée «*test d'occurrence*» interdit d'effectuer des boucles dans les termes comme avec la substitution $\{x \leftarrow f(x)\}$ où `x` est une variable et `f/1` un constructeur d'arité 1.

On peut *appliquer* une substitution $\{x_1 \leftarrow t_1, x_2 \leftarrow t_2, \ldots, x_n \leftarrow t_n\}$ à un terme `T`, ce que l'on note $\{x_1 \leftarrow t_1, x_2 \leftarrow t_2, \ldots, x_n \leftarrow t_n\}$`(T)`, le résultat est obtenu en remplaçant au sein de `T` chaque occurrence de x_i par t_i.

Voici un exemple $\{X \leftarrow pierre\}(pere(jean,X)) = pere(jean,pierre)$

La substitution vide $\sigma = \{\ \}$ est la substitution identité telle que $\forall T(\sigma(T) = T)$

4.2.4 Relation d'ordre partiel entre les termes

On peut définir une relation d'ordre partiel entre les termes. Un terme `T1` *est plus général qu'un* terme `T2` s'il existe une substitution σ telle que σ appliquée à `T1` produit le terme `T2`. Ce que l'on note σ`(T1)=T2`.

On dit aussi que `T2` *est une instance de* `T1`.

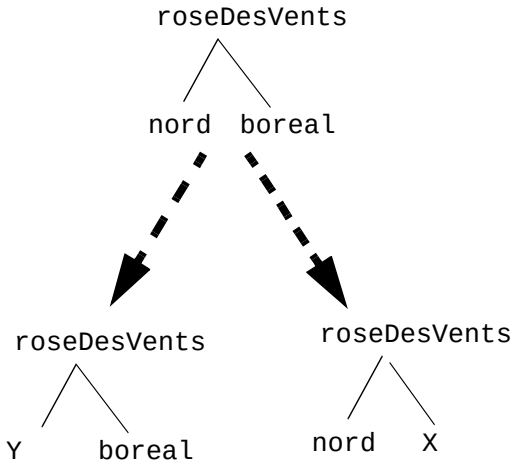
La relation d'ordre est partielle car une telle substitution n'existe pas toujours. Par exemple pour `(homme X)` et `(roseDesVents Y boreal)` on ne peut pas trouver une substitution qui appliquée à l'un des deux termes fait que celui-ci devienne une instance de l'autre.

4.2.5 L'unification dans les langages de programmation logiques

On définit l'unification ainsi :

«*Deux termes T1 et T2 sont unifiables s'il existe un troisième terme T qui est une instance des deux termes T1 et T2*». i.e. ssi $(\exists\sigma)(\exists T)((T = \sigma(T1)) \wedge (T = \sigma(T2)))$

Voici un exemple :



$\sigma = \{(X \leftarrow boreal\ ),(Y \leftarrow nord)\}$

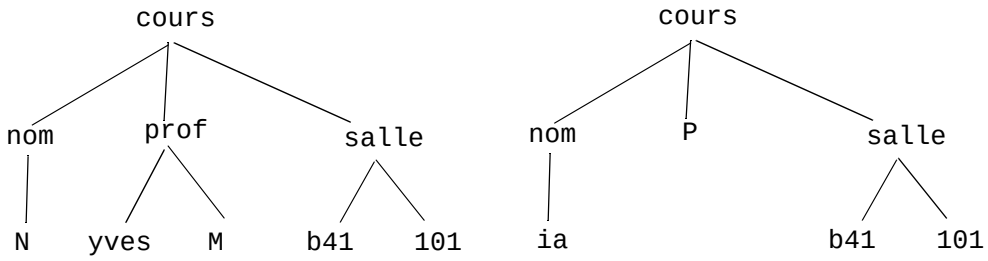
L'unification peut réussir ou échouer, voici des exemples :

TABLEAU 17. Exemples de résultats d'unification

Terme1	**Terme2**	**Résultat**
estUnHomme(socrate)	*estUnHomme(X)*	{X ← socrates}
roseDesVents(sud,austral)	*roseDesVents(X,austral)*	{X ← sud}
roseDesVents(sud,austral)	*roseDesVents(sud,X)*	X ← austral
roseDesVents(sud,austral)	*roseDesVents(X,Y)*	{X ← sud,Y ← austral}
roseDesVents(sud,austral)	*homme(X)*	échec
roseDesVents(sud,austral)	*roseDesVents(sud,boreal)*	échec

Pour deux termes donnés il peut y avoir plusieurs substitutions qui les rendent égaux. Voici un exemple :

```
(cours (nom N) (prof yves M) (salle b41 101))
(cours (nom ia) P (salle b41 101))
```



Voici des substitutions qui rendent les deux termes égaux.

$\sigma_1 = \{N \leftarrow ia, P \leftarrow prof(yves,M)\}$

$\sigma_2 = \{N \leftarrow ia, P \leftarrow prof(yves,marie)\}$

$\sigma_3 = \{N \leftarrow ia, P \leftarrow prof(yves,thomas)\}$

…

4.2.6 Propriété de l'unification des termes du premier ordre

Le problème de l'unification des termes du premier ordre est décidable. De plus, s'il y a des solutions, il en existe une plus générale que toutes les autres (au nom des variables prêt). On dit qu'elle *subsume* toutes les autres. On appelle cette solution le *plus grand unifieur, pgu*.

Pour les deux termes :

```
(cours (nom N) (prof yves M) (salle b41 101))
(cours (nom ia) P (salle b41 101))
```

voici un exemple de plus grand unifieur et deux autres qui sont des instances de celui-ci :

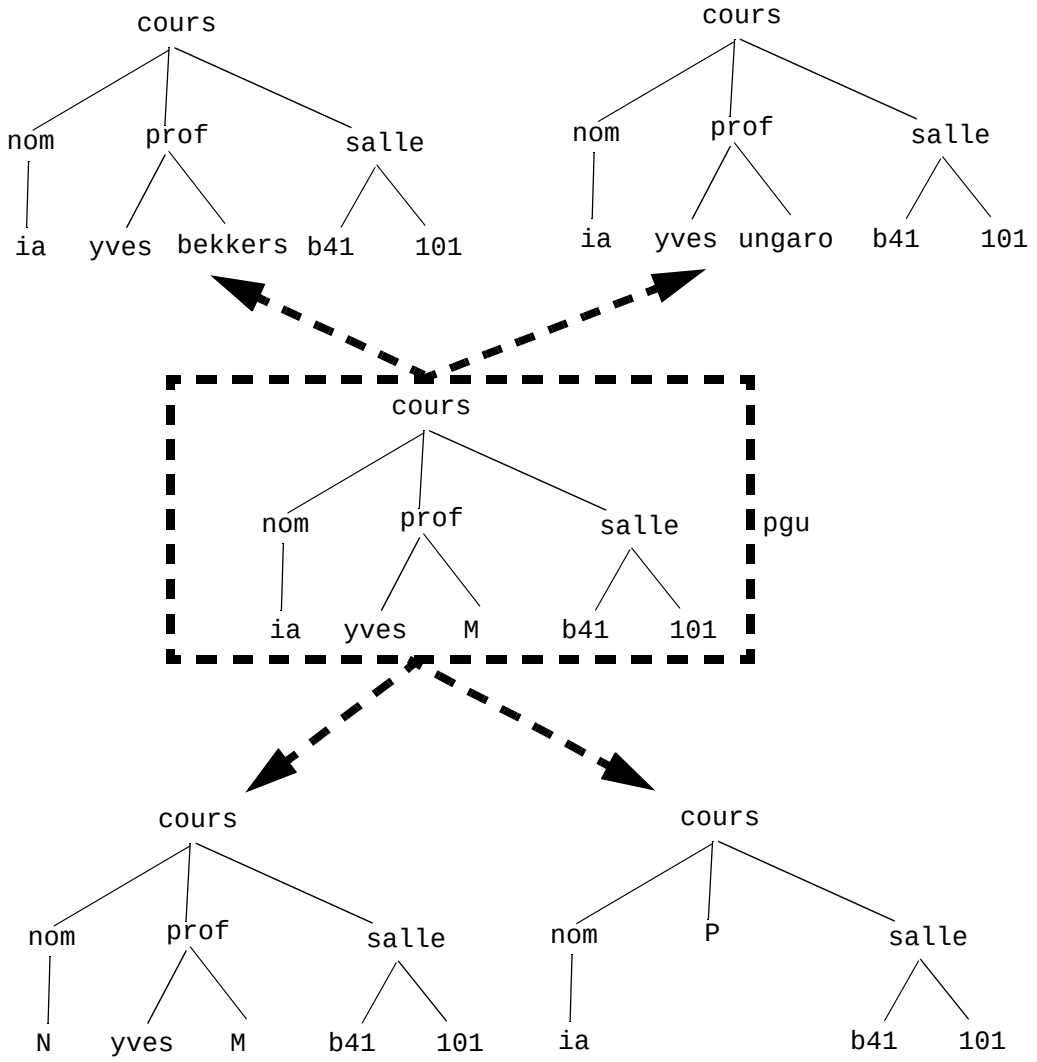


FIGURE 6. Exemple de pgu

4.2.7 Le filtrage dans les langages fonctionnels

Le *filtrage* des langages fonctionnels est une version appauvrie de l'unification qui utilise la relation d'ordre que nous venons de définir. Filtrer un

terme T1 par un terme T2 c'est montrer que T1 est une instance de T2, c'est à dire $(\exists\sigma)(T1 = \sigma(T2))$

4.2.8 Mise en oeuvre de l'unification

Considérons U l'ensemble des variables, C_0 l'ensemble des atomes, C_n $n \in N \wedge n > 0$ les ensembles de constructeurs, $\varnothing$ la substitution identité {}, $\otimes$ la composition de substitutions, voici une proposition d'algorithme d'unification :

```
unifier(t1,t2) {
   if (t1 ou t2 ∈ U) {
      let x la variable;
      let t l'autre terme;
      if (x=t)
         return ⟨true,∅⟩ ;
      else
         if (occurs(x,t)
            return ⟨false,∅⟩ ;
         else
            return ⟨true,{x ← t}⟩
   } else if (t1 ou t2 ∈ C0) {
      let a l'atome;
      let t l'autre terme;
      if (a=t)
         return ⟨true,∅⟩ ;
      else
         return ⟨false,∅⟩ ;
   } else {
      let t1=f(x1, ..., xn) with f ∈ Cn;
      let t2=g(y1, ..., ym) with g ∈ Cm;
      if (f≠g)
         return ⟨false,∅⟩ ;
      else (f=g ∧ n=m=k) {
         boolean ok=true;
         substitution σ=∅;
         for (int i=1; i<k+1; i++) {
              ⟨b,σ1⟩ = unifier(σ(xi), σ(yi));
            if (not(b))
               return ⟨false,∅⟩ ;
            else
               σ = σ ⊗ σ1
            }
            return ⟨true,σ⟩
         }
      }
   }
```

4.2.9 Généralisation de la règle d'effacement

Nous pouvons maintenant énoncer une généralisation de la règle d'effacement d'un but par l'interpréteur λPROLOG :

Règle d'effacement généralisée :

«On dit qu'un but `B` s'efface avec succès dans un programme `P` s'il existe un fait F dans le programme `P` qui s'unifie avec `B` selon la substitution la plus générale σ.»

Ce qui se représente par le schéma suivant :

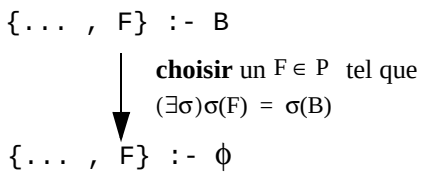


FIGURE 7. Règle d'effacement

FIGURE 8. Effacement d'un but après unification du but avec un fait du programme

Voici donc la démonstration par λPROLOG dans notre programme de la «*rose des vents*» qu'il existe une direction correspondant à l'adjectif austral :

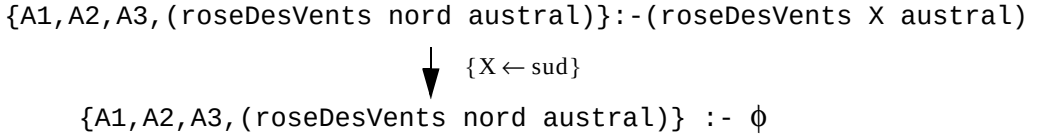


FIGURE 9. Effacement du but `:-(roseDesVents X austral)`

4.2.10 Non déterminisme, stratégie ou

Voici un énoncé :

«existe-t-il une direction D et un adjectif A tels que A est l'adjectif correspondant à la direction D».

Sa représentation en logique est la suivante :

$(\exists x)(\exists y)roseDeVents(x,y)$

Elle se représente concrètement en λPROLOG par la question composée d'un simple but :

```
:-(roseDesVents X Y).
```

Ce terme peut s'unifier successivement avec les quatre faits du programme rose des vents. Les substitutions correspondantes des variables `X` et `Y` sont :

$\{X \leftarrow sud, Y \leftarrow austral\}$
$\{X \leftarrow nord, Y \leftarrow boreal\}$
$\{X \leftarrow est, Y \leftarrow oriental\}$
$\{X \leftarrow ouest, Y \leftarrow occidental\}$

Tant qu'il n'a pas effacé son but λPROLOG explore tous les faits du programme de manière séquentielle jusqu'à obtention d'un effacement. Si le premier fait ne convient pas, il en essaye un autre, etc. L'interpréteur essaye avec

tous les faits du programme. On appelle *stratégie* OU la stratégie d'exploration adoptée par l'interpréteur pour parcourir les faits du programme.

Stratégie OU **de** λPROLOG**.** La stratégie OU de λPROLOG consiste à explorer les faits en séquence de haut en bas, dans l'ordre où ils se trouvent dans le programme.

L'interpréteur λPROLOG est *paresseux* en ce sens qu'il s'arrête à la première solution qu'il trouve. Cependant si l'utilisateur désire obtenir toutes les solutions d'un but, il lui suffit de faire croire à l'interpréteur, dès qu'il a mis une solution en évidence, qu'il est en échec. De cette manière il repart à la recherche de la prochaine solution. Et c'est alors la stratégie OU qui décide alors de l'ordre dans lequel les solutions (les témoins) sont produits.

Lorsque l'on pose une question à λPROLOG on ne sait pas s'il va y avoir zéro, une ou plusieurs réponses. La programmation logique s'oppose ainsi à la programmation fonctionnelle où une expression se réduit à un et un seul résultat. On dit parfois que les langages de programmation logique *sont non-déterministes*. Non pas parce qu'ils répondent de manière aléatoire mais parce qu'on ne connaît pas à priori le nombre de réponses que l'on va obtenir.

4.2.11 Sens des paramètres d'un prédicat : notion de mode d'un prédicat

On constate que le même prédicat est utilisé pour résoudre de nombreuses questions différentes. Cette propriété est à l'image de ce qui se passe lorsque l'on interroge une base de données relationnelle où la même table peut donner lieu à de nombreuses questions.

Cette polyvalence d'utilisation des prédicats est une des originalités de la programmation logique. Les fonctions utilisées dans OCAML, du fait de leur nature même, n'ont pas cette capacité de mode multiple d'utilisation :

TABLEAU 18. Des succès des échecs

Enoncée de la question	**But λ*PROLOG* correspondant**	**Résultat**
Est-ce que l'adjectif boreal correspond à la direction nord ?	`(roseDesVents nord boreal)`	succès
Est-ce que l'adjectif austral correspond à la direction nord ?	`(roseDesVents nord austral)`	échec
Quel est l'adjectif correspondant à la direction nord ?	`(roseDesVents nord X)`	{ X ← boreal }

TABLEAU 18. Des succès des échecs

Enoncée de la question	But λ*PROLOG* correspondant	Résultat
Quelle est la direction correspondant à l'adjectif austral ?	`(roseDesVents X austral)`	{X ← sud}
Donnez les direction et adjectifs qui se correspondent.	`(roseDesVents X Y)`	{X ← sud,Y ← austral} {X ← nord,Y ← boreal } {X ← est,Y ← oriental} {X ← ouest,Y ← occidental}

Le mode d'exécution d'un prédicat peut être spécifié à l'aide des trois symboles suivants «+», «-» et «?» placés devant l'identifiant de paramètre. Voici des exemples de déclaration de modes pour le prédicat binaire `roseDesVents` :

TABLEAU 19. Modes d'un prédicat

Déclaration de mode	Signification
`(roseDesVents +D +A)`	Utilisation en vérification : La direction D et l'Adjectif A sont instanciées au moment de l'appel du prédicat.
`(roseDesVents +D -A)`	Calcul d'un adjectif à partir d'une direction donnée D, A est une variable,
`(roseDesVents -D +A)`	Calcul d'une direction à partir d'un adjectif donné A, D est une variable
`(roseDesVents -D -A)`	Générer sous forme d'un flot de résultats (en utilisant le non-déterminisme de λPROLOG) tous les couples `<A, D>`, à l'appel `A` et `D` sont toutes les deux des variables.

- \+ devant in identificateur signifie que le paramètre est en entré et qu'il doit donc être complètement instancié au moment de l'appel
- \- devant un identificateur signifie que le paramètre est en sortie et qu'il doit être une variable libre au moment de l'appel.
- ? devant un identificateur signifie que qu'il peut être indifféremment libre ou lié à l'appel

4.3 Des variables dans les faits, règle d'instanciation

Considérons la propriété suivante de l'addition : «*Zero est l'élément neutre de l'addition*». Cette propriété est modélisée en logique par l'axiome :

$(\forall x)$somme(zero,x,x)

λPROLOG admet les variables dans les faits. L'axiome ci-dessus est représenté par le fait :

```
somme zero X X.
```

Comme pour les buts, la quantification est omise en programmation logique mais ici il est toujours supposé universel. Voici en résumé les formules dont nous disposons maintenant dans notre reconstruction de PROLOG :

TABLEAU 20. Reconstruction de PROLOG étape 1

	Représentation en λPROLOG	Représentation en logique
Fait sans variable	`(homme socrate)`	homme(socrate)
but sans variable	`:-(homme socrate)`	homme(socrate)
Fait ou axiome quantifiés	`(homme X)`	(∀x)homme(x)
Buts ou question quantifié	`:-(homme X)`	(∃x)homme(x)

4.3.1 Point de vue logique

La déduction naturelle nous offre une troisième règle appelée règle d'instanciation qui s'énonce comme suit : «*un but complètement instancié peut être inféré d'un but ressemblant quantifié universellement*». La règle s'écrit :

$$\frac{(\forall x)\text{somme(zero,x,x)}}{\text{somme(zero,un,un)}}\,INS$$

FIGURE 10. Règle d'instanciation

4.3.2 Point de vue opérationnel

Du point de vue opérationnel, la règle d'effacement définie plus haut s'applique ici à nouveau. Ainsi nous pouvons écrire l'effacement suivant :

`{(somme zero X X)}:-(somme zero un un)`

$\downarrow \{X \leftarrow un\}$

`{(somme zero X X)} :-` ϕ

FIGURE 11. Effacer le but `:-(somme zero un un)`

4.3.3 Combiner les règles d'instanciation/généralisation

Du point de vue logique, nous pouvons combiner les règles d'instanciation et de généralisation pour effectuer la démonstration suivante :

$$\dfrac{\dfrac{\dfrac{}{(\forall x)\text{somme(zero,x,x)}}\,A}{\text{somme(zero,un,un)}}\,INS}{(\exists y)\text{somme(zero,un,Y)}}\,GEN$$

FIGURE 12. Démontrer (∃y)somme(zero,un,Y)

Du point de vue opérationnel, cette démonstration s'explique par la même règle d'effacement que celle définie plus haut :

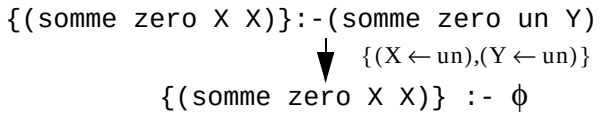


FIGURE 13. Effacer le but `:-(somme zero un Y)`

4.4 Où les axiomes deviennent des règles d'inférence

Les formules dont nous disposons pour l'instant dans notre reconstruction sont limitées aux formules atomiques ou aux formules atomiques quantifiées, voir Figure 20, "Reconstruction de PROLOG étape 1", page 25. Dans ce qui suit nous étendons le langage des axiomes.

4.4.1 Règle de Modus Ponens simplifiée.

La dernière règle de la déduction naturelle que nous présentons est la règle de *Modus Ponens*. Un énoncé classique de cette règle est :

> «*Socrate est un homme et tous les hommes sont mortels donc socrate est mortel.*»

Cet énoncé comporte deux prémisses qui s'expriment en logique par la conjonction des deux formules suivantes

$$\left\{\begin{array}{c} \text{homme(socrates)} \\ (\forall x)(\text{mortel(x)} \Leftarrow \text{homme(x)}) \end{array}\right\}$$

FIGURE 14. Théorie «*Socrate est mortel*»

Nous appelons cette théorie «*socrate est un mortel*».

Soit `H` et `A` des formules atomiques. Voici une première version simplifiée de la règle de Modus Ponens où `H`, `A` et `M` sont des formules atomiques :

$$\frac{\text{H} \qquad \text{A} \Leftarrow \text{H}}{\text{A}} MP$$

FIGURE 15. Modus ponens simplifié

Cette règle s'énonce aisément comme suit :

> «*si* `H` *et* `H => A` *sont deux axiomes de ma théorie alors* `A` *est un théorème de ma théorie*».

Il est maintenant possible de démontrer que *mortel(socrate)* est un théorème de la théorie «*socrate est un mortel*»

Voici la courte démonstration :

$$\dfrac{\dfrac{}{\text{homme(socrates)}}A \qquad \dfrac{\dfrac{}{(\forall x)(\text{mortel(x)} \Leftarrow \text{homme(x)})}A}{\text{mortel(socrates)} \Leftarrow \text{homme(socrates)}}INS}{\text{mortel(socrates)}}MP$$

FIGURE 16. Démontrer en logique que Socrate est mortel

Voici encore une courte démonstration dans la même théorie du théorème : $(\exists x)\text{mortel(x)}$. Démonstration

$$\dfrac{\dfrac{\dfrac{}{\text{homme(socrates)}}A \qquad \dfrac{\dfrac{}{(\forall x)(\text{mortel(x)} \Leftarrow \text{homme(x)})}A}{\text{mortel(socrates)} \Leftarrow \text{homme(socrates)}}INS}{\text{mortel(socrates)}}MP}{(\exists x)\text{mortel(x)}}GEN$$

FIGURE 17. Démontrer en logique qu'il existe un mortel

4.4.2 Des formules complexes dans le programme

Pour représenter dans les programmes une formule complexe avec implication telle que $(\forall x)(\text{mortel(x)} \Leftarrow \text{homme(x)})$ λPROLOG introduit l'opérateur binaire infixe implique qui s'écrit concrètement `:-` et qui signifie l'implication à gauche $\Leftarrow$.

Les deux axiomes $\left\{\begin{array}{c}\text{homme(socrates)}\\(\forall x)(\text{mortel(x)} \Leftarrow \text{homme(x)})\end{array}\right\}$ de la théorie «*Socrate est mortel*» s'écrivent en λPROLOG :

```
est_un_homme socrate.
est_mortel X :- est_un_homme X.
```

En PROLOG le second axiome s'appelle une *clause* ou *règle* pour le différencier des faits qui sont des formules atomiques.

La partie à droite de l'opérateur :- est appelé le *corps* de la clause. La partie gauche est appelée la *tête* de la clause.

Le type de l'opérateur `(:-)` en `λProlog` est définit comme suit :

```
kind o type.
type (:-) o -> o -> o.
```

Le type de l'expression

```
est_mortel X :- est_un_homme X.
```

est o. Elle pourrait être écrite de manière préfixe comme suit :

```
(:-) (est_mortel X) (est_un_homme X).
```

λPROLOG admet les deux écritures.

4.4.3 Ordre supérieur

λPROLOG permet de manipuler des termes d'ordre supérieur. Par exemple il est possible d'écrire l'égalité suivante :

```
F = ((:-) (est_mortel X))
```

La variable `F` reçoit une valeur de type `o -> o`. Il s'agit d'un «*prédicat incomplet*». Formellement cette construction est une fonction des prédicats dans les prédicats. Ces constructions qui sont dites d'ordre supérieur sont des objets à part entière du langage λPROLOG au même titre que les prédicats o, les entiers ou les string. Dans le contexte de la portée de `X` on peut appliquer ultérieurement la variable `F` à un prédicat comme dans `(F (est_un_homme X)),` on obtient le prédicat :

```
(:-) (est_mortel X) (est_un_homme X).
```

4.4.4 Portée des variables

La quantification de la variable `X` porte sur toute la clause et seulement la clause. C'est-à-dire qu'une autre variable `X` dans une autre clause ou un autre fait du programme est différente de la première.

Voici le programme λPROLOG pour représenter la théorie «*les hommes sont mortels et socrate est un homme*». :

```
kind homme type.
type socrate homme.

type est_un_homme homme -> o.
est_un_homme socrate.

type est_mortel homme -> o.
est_mortel X :- est_un_homme X.
```

4.4.5 Comment λPROLOG se débrouille opérationnellement avec Modus Ponens ?

La règle d'effacement de but que nous avons utilisé jusqu'à présent ne suffit plus pour expliquer comment λPROLOG modélise les démonstrations effectuées en déduction naturelle grâce à la règle Modus Ponens.

Nous avons besoin d'une nouvelle règle appelée *règle de remplacement* qui se représente comme suit :

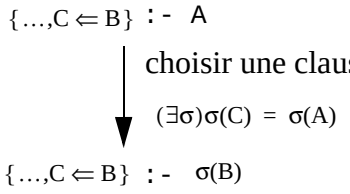


FIGURE 18. Règle de remplacement

Cette règle s'énonce ainsi

«*Pour éliminer un but* A *dans un programme* P *qui contient une clause* $C \Leftarrow B$ *telle que* C *la tête de clause s'unifie avec le but* A *selon une substitution* σ, c'est à dire $\sigma(A) = \sigma(C)$, *il suffit de remplacer le but* A *par* $\sigma(B)$ *dans le programme* P.»

On remplace ainsi un problème par un autre problème que l'on essaye de résoudre à son tour en éliminant le nouveau but dans lequel on a cumulé la substitution obtenue dans l'étape précédente.

Soit le programme `P` suivant :

```
P = {homme socrate, mortel X :- homme X}
```

Voici un exemple de d'effacement de but pour la question `:- (mortel X)` dans ce programme :

```
{homme socrate, mortel X :- homme X} :- (mortel M)
                        ↓ {X1 <- M}
{homme socrate, mortel X :- homme X} :- (homme M)
                        ↓ {M <- socrate}
{homme socrate, mortel X :- homme X} :- ϕ
```

FIGURE 19. Effacement du but `:- (mortel X)` par Modus Ponens

4.4.6 Définir un prédicat à l'aide plusieurs clauses

De même que l'on peut définir un prédicat à l'aide plusieurs faits, il est possible de définir un prédicat à l'aide de plusieurs clauses. Supposons la théorie composée des quatre axiomes suivants

$$\left\{\begin{array}{c} \text{mere(marie,jean)} \\ \text{pere(pierre,jean)} \\ (\forall x)(\forall y)(\text{parent}(x,y) \Leftarrow \text{mere}(x,y)) \\ (\forall x)(\forall y)(\text{parent}(x,y) \Leftarrow \text{pere}(x,y)) \end{array}\right\}$$

FIGURE 20. Théorie «*parent*»

Cette théorie définit le prédicat binaire *parent* à partir des deux prédicats binaires *mere* et *pere*. On définit ici une relation comme l'union de deux relations.

En λPROLOG cette théorie se définit comme suit :

```
kind personne type.
type (marie, jean, pierre) personne.

type (pere, mere, parent) personne -> personne -> o.

pere pierre jean.
mere marie jean.

parent X Y :- mere X Y.
parent X Y :- pere X Y.
```

Notez bien, comme nous l'avons déjà dit, qu'en λPROLOG les variables X et Y des deux clauses sont locales à chacune des clauses.

Exercice 1.

Ecrire en déduction naturelle deux démonstrations du théorème $(\exists x)\text{parent}(x,\text{jean})$ *dans la théorie* «parent» *ci-dessus.*

Exercice 2.

*Faite les deux mêmes démonstrations avec le démonstrateur de λ*PROLOG *et le programme ci dessus.*

4.5 Où les buts deviennent des formules complexes

Nous introduisons maintenant le connecteur logique $\wedge$ dans les questions.

4.5.1 Point de vue logique

Supposons nous ayons une théorie que nous appellerons «*marie*» qui est composée des deux axiomes suivants $\{\text{femme}(\text{marie}),\text{parent}(\text{pierre},\text{marie})\}$.

Le théorème que nous voulons prouver dans cette théorie est le suivant : $\text{femme}(\text{marie}) \wedge \text{parent}(\text{pierre},\text{marie})$

La déduction naturelle permet d'introduire le connecteur logique $\wedge$ entre deux formules si les deux formules sont des théorèmes. Cette règle dite «*règle d'introduction du* $\wedge$» s'écrit visuellement comme suit :

$$\frac{A \qquad B}{A \wedge B} et_I$$

FIGURE 21. Règle d'introduction du ET

Ces conjonctions de formules peuvent être quantifiées existentiellement. Par exemple nous pouvons exprimer la question :

«existe t-il une personne qui est le parent d'une femme qui existe ?»

ce qui s'écrit en logique $(\exists x)\exists y(\text{femme}(x) \wedge \text{parent}(y,x))$

Voici une démonstration en déduction naturelle du théorème $(\exists x)\exists y(\text{femme}(x) \wedge \text{parent}(y,x))$ dans la théorie «*marie*» :

$$\dfrac{\dfrac{\dfrac{}{\text{femme}(\text{marie})}A \qquad \dfrac{}{\text{parent}(\text{pierre},\text{marie})}A}{\text{femme}(\text{marie}) \wedge \text{parent}(\text{pierre},\text{marie})}et_I}{(\exists y)(\text{femme}(\text{marie}) \wedge \text{parent}(y,\text{marie}))}gen}{(\exists x)(\exists y)(\text{femme}(x) \wedge \text{parent}(y,x))}gen$$

FIGURE 22. Démonstration en logique du théorème $(\exists x)\exists y(\text{femme}(x) \wedge \text{parent}(y,x))$

4.5.2 Nouvelle extension du langage PROLOG

Considérons la théorie {femme(marie),parent(pierre,marie)}. Nous la représentons en λProlog par le programme «*marie*» suivant :

```
kind personne type.
type (marie, pierre) personne.

type femme personne -> o.
femme marie.

type parent personne -> personne -> o.
parent pierre marie.
```

Pour représenter une question construite telle que $(\exists x)\exists y(femme(x) \wedge parent(y,x))$ λProlog introduit le connecteur ET dans sa syntaxe des questions. Ce connecteur s'écrit (,). Son type en λProlog est :

```
type (,) o -> o -> o.
```

La formule $(\exists x)\exists y(femme(x) \wedge parent(y,x))$ se représente en λProlog par :

```
:- femme X, parent Y X.
```

4.5.3 Nouvelle version des règles d'interprétation de Prolog

Voici une nouvelle version des deux règles d'interprétation de Prolog. Elles prennent en compte le connecteur ∧ dans les questions :

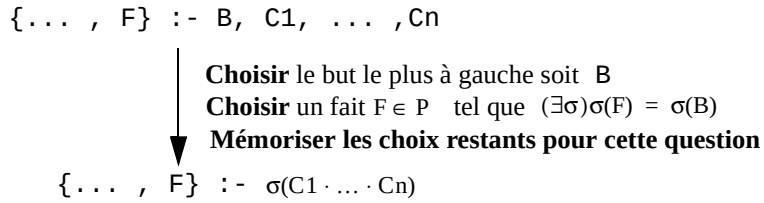


FIGURE 23. Règle d'effacement d'un but

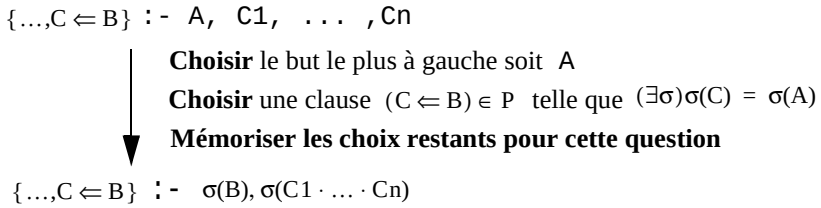


FIGURE 24. Règle de remplacement

Voici un exemple de d'effacement de but pour la question

```
:- (femme X, parent Y X)
```

dans le programme «*marie*» donné ci dessus :

```
{femme marie,
parent pierre marie} :- femme X, parent Y X.
                     ↓ {X <- marie}
{femme marie,
parent pierre marie}  :- parent Y marie.
                     ↓ {Y <- pierre}
{femme marie,
parent pierre marie} :- ϕ
```

FIGURE 25. Effacement du but `:-(femme X, parent Y X)`

4.6 Où les corps des clauses deviennent des formules complexes

Considérons la formulation logique du prédicat binaire *aPourFille* suivante :

$$(\forall x)(\forall y)(\text{aPourFille}(x,y) \Leftarrow \text{parent}(x,y) \wedge \text{femme}(y))$$

Dans notre reconstruction actuelle le corps des clauses était constitué d'une formule atomique. En fait, λPROLOG utilise la même syntaxe pour le corps des clauses que pour les questions complexes que nous venons de voir. Un corps de clause peut être construit avec le connecteur logique $\wedge$, que l'on écrit concrètement (,).

Considérons une théorie composée des axiomes suivants :

$$\left\{ \begin{array}{c} \text{femme(marie)} \\ \text{parent(pierre,marie)} \\ (\forall x)(\forall y)(\text{aPourFille}(x,y) \Leftarrow \text{parent}(x,y) \wedge \text{femme}(y)) \end{array} \right\}$$

On peut démontrer en déduction naturelle le théorème suivant de cette théorie : $(\exists x)(\exists y)\text{aPourFille}(x,y)$. Voici la démonstration :

$$\dfrac{\dfrac{\dfrac{}{\text{parent(pierre,marie)}}A \quad \dfrac{}{\text{femme(marie)}}A \quad \dfrac{\dfrac{}{(\forall x)(\forall y)(\text{aPourFille}(x,y) \Leftarrow \text{parent}(x,y) \wedge \text{femme}(y))}A}{\text{aPourFille(pierre,marie)} \Leftarrow \text{parent(pierre,marie)} \wedge \text{femme(marie)}}INS}{\text{aPourFille(pierre,marie)}}MP}{(\exists x)(\exists y)\text{aPourFille}(x,y)}GEN$$

Voici le programme `P` représentant cette théorie :

```
kind personne type.
type (marie, pierre) personne.

type femme personne -> o.
femme marie.             /* (1) */

type parent personne -> personne -> o.
parent pierre marie.     /* (2) */

type aPourFille personne -> personne -> o.

aPourFille X Y :-        /* (3) */
   parent X Y,
   femme Y.
```

Voici la démonstration faite ci-dessus effectuée par l'interpréteur λPROLOG :

```
P :- aPourFille X Y.
    ↓(3) {X1 ← X,Y1 ← Y}
P :- parent X Y, femme Y.
    ↓(2) {X ← pierre,Y ← marie}
P :- femme marie.
    ↓(1) ∅
P :- ∅
```

5 Déduction naturelle et Prolog - vue générale

Avant de récapituler par une vue générale la reconstruction que nous venons de faire, nous introduisons les connecteurs $\exists$ et $\vee$ dans les questions et les parties droites de règle.

5.1 Le connecteur logique $\exists$ dans les parties droites de règle

La logique nous dit que la formule suivante est une tautologie (i.e. toujours vraie) :

$$(\forall x)(\forall y)(P(x) \Leftarrow Q(x,y)) \Leftrightarrow (\forall x)(P(x) \Leftarrow (\exists y)Q(x,y))$$

C'est à dire que les deux formules $(\forall x)(\forall y)(P(x) \Leftarrow Q(x,y))$ et $(\forall x)(P(x) \Leftarrow (\exists y)Q(x,y))$ sont équivalentes et peuvent être remplacées l'une par l'autre. Ainsi dans une théorie de la famille nous pourrions énoncer l'axiome suivant :

«*Pour être une mère il suffit d'être la mère de quelqu'un*»

Ce qui s'énonce en logique par la formule : $(\forall x)(\text{estMere}(x) \Leftarrow (\exists y)\text{mere}(x,y))$. Dans la syntaxe de PROLOG que nous avons donné jusqu'à maintenant cet axiome ne peut en principe pas s'écrire, mais on peut le remplacer par :

```
estMere X :- mere X Y.
```

Ce qui signifie formellement en logique $(\forall x)(\forall y)(\text{estMere}(x) \Leftarrow \text{mere}(x,y))$ et qui est équivalente à la formule logique précédente.

Exercice 3.

Soit la théorie «marieEstUneMere» *composée des deux axiomes suivants :*

$$\left\{ \begin{array}{c} \text{mere(marie,pierre)} \\ (\forall x)(\forall y)(\text{estMere}(x) \Leftarrow \text{mere}(x,y)) \end{array} \right\}$$

Démontrez en déduction naturelle que la formule estMere(marie) *est un théorème de cette théorie.*

Représentez en λPROLOG la théorie «marieEstUneMere» *et développez l'exécution du but correspondant à la formule* estMere(marie).

Exercice 4.

Même exercice que précédemment mais avec la théorie :

$$\left\{\begin{array}{c} \text{mere(marie,pierre)} \\ (\forall x)(\text{estMere(x)} \Leftarrow (\exists y)\text{mere(x,y)}) \end{array}\right\}$$

En conclusion PROLOG ne permet pas d'écrire des axiomes tels que $(\forall x)(P(x) \Leftarrow (\exists y)Q(x,y))$ mais le programmeur peut toujours réécrire ce type d'axiome sous la seconde forme $(\forall x)(\forall y)(P(x) \Leftarrow Q(x,y))$ et effectuer ainsi les mêmes démonstrations.

Nous verrons que seul λPROLOG, plus complet, implémente le connecteur logique $\exists$ dans la partie droite des règles.

5.2 Connecteur $\vee$ dans la partie droite des règles ou dans les questions.

Le dernier type de règle que nous voyons sont les implications comportant une disjonction en partie droite de règle. Voici un exemple :
$(\forall \vec{x})(P(\vec{x}) \Leftarrow Q_1(\vec{x}) \vee Q_2(\vec{x}))$.

La logique nous apprend que la formule suivante

$$((\forall \vec{x})(P(\vec{x}) \Leftarrow Q_1(\vec{x}) \vee Q_2(\vec{x}))) \Leftrightarrow ((\forall \vec{x})(P(\vec{x}) \Leftarrow Q_1(\vec{x})) \wedge (\forall \vec{x})(P(\vec{x}) \Leftarrow Q_2(\vec{x})))$$

est une tautologie. On peut donc toujours remplacer un axiome de la forme $(\forall \vec{x})(P(\vec{x}) \Leftarrow Q_1(\vec{x}) \vee Q_2(\vec{x}))$ par les deux axiomes $\{(\forall \vec{x})(P(\vec{x}) \Leftarrow Q_1(\vec{x})),(\forall \vec{x})(P(\vec{x}) \Leftarrow Q_2(\vec{x}))\}$ et réciproquement.

Voici un exemple. L'axiome :

$$(\forall x)(\forall y)(\text{parent(x,y)} \Leftarrow \text{pere(x,y)} \vee \text{mere(x,y)})$$

peut être remplacé par les deux axiomes :

$$\left\{\begin{array}{l} (\forall x)(\forall y)(\text{parent(x,y)} \Leftarrow \text{pere(x,y)}) \\ (\forall x)(\forall y)(\text{parent(x,y)} \Leftarrow \text{mere(x,y)}) \end{array}\right\}$$

5.2.1 Point de vue de la déduction naturelle

La déduction naturelle propose deux règles pour introduire une disjonction $P \vee Q$:

$$\frac{P}{P \vee Q} ou_I \qquad \frac{Q}{P \vee Q} ou_I$$

C'est à dire que l'on peut introduire une disjonction $P \vee Q$ dès que l'on a démontré P ou que l'on a démontré Q.

Exercice 5.

Considérez la théorie composée des axiomes suivants :

$$\left\{\begin{array}{c} \text{pere(pierre,marie)} \\ \text{mere(jeanne,marie)} \\ (\forall x)(\forall y)(\text{parent}(x,y) \Leftarrow \text{pere}(x,y) \vee \text{mere}(x,y)) \end{array}\right\}$$

Démontrez en déduction naturelle de deux manières le théorème suivant de cette théorie $(\exists x)\text{parent}(x,\text{marie})$

5.2.2 Point de vue opératoire

Tous les langages de programmation logiques tels que PROLOG et λPROLOG implémentent le connecteur ∨ dans la partie droite des règles. Concrètement le connecteur ∨ s'écrit (;). Le type de cet opérateur est :

```
type (;) o -> o -> o.
```

Voici un tableau résumant le type des connecteurs logiques binaires de λPROLOG que nous avons vu jusqu'à présent :.

TABLEAU 21. Principaux opérateurs logiques binaire de λProlog

logique	λProlog	Définition du type
⇐	`:-`	`type (:-) o -> o -> o.`
∧	`,`	`type (,) o -> o -> o.`
∨	`;`	`type (;) o -> o -> o.`

Nous devons introduire la nouvelle règle suivante qui interprète une disjonction dans la question :

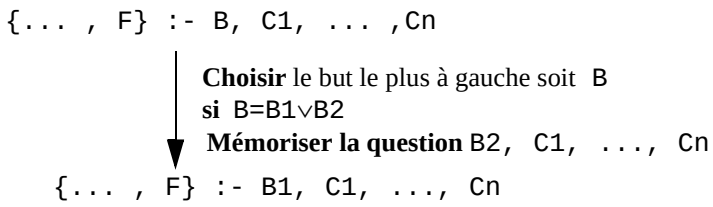


FIGURE 26. Règle d'interprétation du ∨

Exercice 6.

Ecrire en λPROLOG le programme de l'Exercice 5, page 35. déroulez l'exécution de la question `:- parent X marie`.

5.3 Vue Générale

Notre définition du langage PROLOG est maintenant complète. Voir la section suivante pour une récapitulation de sa définition complète. Voici les versions définitives de nos règles pour les démonstrations en déduction naturelle et pour le fonctionnement d'un interpréteur PROLOG.

5.3.1 Déduction naturelle

Versions définitives des règles de déduction naturelle adaptées à PROLOG :

$$\frac{}{A}\,A \qquad \frac{A \quad B}{A \wedge B}\,et_I \qquad \frac{(\forall x)P(x)}{P(cste)}\,INS$$

Règle d'identité ou axiome — **Règle d'introduction du $\wedge$** — **Règle d'instanciation ou élimination du $\forall$**

$$\frac{P(cste)}{(\exists x)P(x)}\,GEN \qquad \frac{H1 \wedge \ldots \wedge Hn \qquad A \Leftarrow H1 \wedge \ldots \wedge Hn}{A}\,MP$$

Règle de généralisation ou introduction du $\exists$ — **Règle de Modus Ponens ou élimination du $\Leftarrow$**

$$\frac{P}{P \vee Q}\,ou_I \qquad \frac{Q}{P \vee Q}\,ou_I$$

Règles d'introduction du $\vee$

FIGURE 27. La déduction naturelle adaptée à PROLOG

Exercice 7.

Considérez la théorie $\left\{ \begin{array}{c} femme(marie) \\ parent(pierre,marie) \\ (\forall x)(\forall y)(aPourFille(x,y) \Leftarrow parent(x,y) \wedge femme(y)) \end{array} \right\}$*. que nous appellerons ici la théorie «aPourFille».*

Démontrez en déduction naturelle que la formule : $aPourFille(pierre,marie)$ *est un théorème de cette théorie.*

Complétez cette démonstration pour démontrer que la formule : $(\exists x)(\exists y)aPourFille(x,y)$ *est aussi un théorème de la même théorie.*

5.3.2 Fonctionnement d'un interpréteur PROLOG

Versions définitives des deux règles définissant le fonctionnement d'un interpréteur Prolog. On prend toujours le premier but le plus à gauche de la question et un lui applique une des trois règles suivantes :

```
{... , F} :- B, C1, ... ,Cn
```

Choisir le but le plus à gauche soit `B`
Choisir un fait $F \in P$ tel que $(\exists\sigma)\sigma(F) = \sigma(B)$
Mémoriser les choix restants pour cette question

```
{... , F} :- σ(C1 · … · Cn)
```

Règle d'effacement

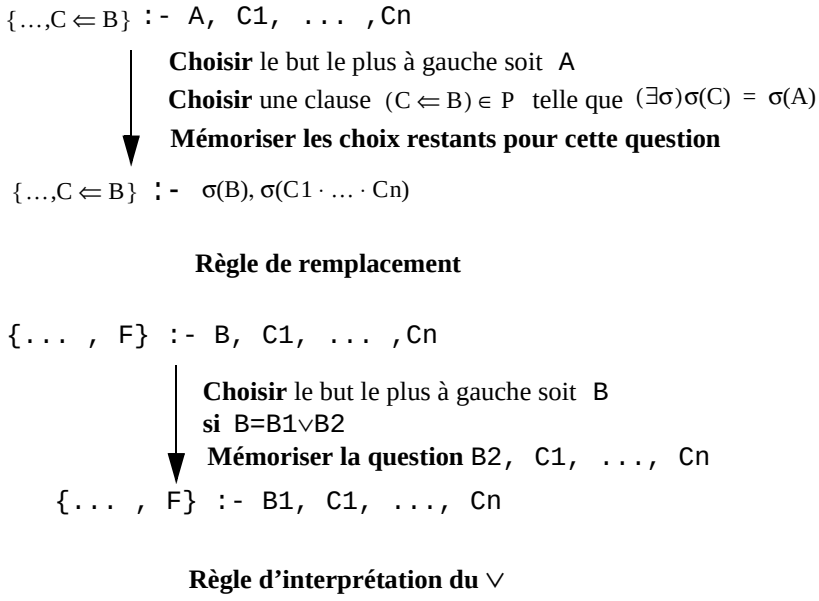


FIGURE 28. Règles de fonctionnement d'un interpréteur Prolog

Exercice 8.

*Représentez en λProlog la théorie «*estLaFilleDe*» ci-dessus. Utilisez l'interpréteur et ses deux règles de fonctionnement pour effectuer deux démonstrations de la question* `:- aPourFille X Y`.

6 Syntaxe de Prolog

Voici un résumé sous forme de tableau de notre reconstruction de Prolog :

TABLEAU 22. Reconstruction de Prolog

Logique		**λProlog**
pere(jean,marie)		`pere jean marie.`
(∀x)somme(zero,x,x)		`somme zero X X.`
(∃x)pere(jean,x)	*?*	`:- pere jean X.`
(∀x)(mortel(x) ⇐ homme(x))		`mortel X :- homme X`
(∃x)∃y(femme(x) ∧ parent(y,x))	*?*	`:- femme X, parent Y X.`
(∀x)(∀y)(aPourFille(x,y) ⇐ parent(x,y) ∧ femme(y))		`aPourFille X Y :-` `  parent X Y, femme Y.`
(∀x)(estMere(x) ⇐ (∃y)mere(x,y))		∅
(∀x)(∀y)(estMere(x) ⇐ mere(x,y))		`estMere X :- mere X Y.`
(∀x)(∀y)(parent(x,y) ⇐ pere(x,y) ∨ mere(x,y))		`parent X Y :-` `  pere X Y; mere X Y.`

Dans ce tableau les formules annotées ? sont des questions, en logique on dirait des théorèmes à prouver. Les autres formules sont des clauses. Chaque clause représente un axiome de la logique.

6.1 Définition

La syntaxe de PROLOG est définie formellement en deux temps.

6.1.1 Les formules atomiques ou termes

Considérons les termes comme définis en “Syntaxe des termes du premier ordre”, page 17.

6.1.2 Les formules non atomiques

Les formules non atomiques sont définies par la double définition suivante :

$$D = A \mid \forall x D \mid A \Leftarrow G$$

$$G = A \mid G \wedge G \mid \exists x G \mid G \vee G$$

L’ensemble de D constitue les *règle*, en logique on les appel des *clauses de Horn* (en anglais *Definite Clauses*). L’ensemble G constitue les *buts* ou *questions* (en anglais *goals*). Un programme PROLOG est un ensemble de règles qui définissent la théorie plus une question qui constitue le théorème à démontrer dans la théorie.

6.2 Syntaxes concrètes - Prolog versus λProlog

Voici comment traduire en Prolog quelques exemples de formules logiques :

TABLEAU 23. Prolog : correspondance avec la logique

Logique	Prolog
chat(felix)	`chat(felix)`
$(\forall x)(\text{ronronne}(x) \Leftarrow \text{chat}(x))$	`ronronne(X) :- chat(X)`
$(\exists x)\text{ronronne}(x)$	`:- ronronne(X)`
pere(felix,noireau)	`pere(felix,noireau)`
$(\forall x)(\text{est} - \text{mere}(x) \Leftarrow (\exists y)\text{mere}(x,y))$	`est-mere(X) :- mere(X,Y)`

Dans la colonne **Prolog** du tableau ci-dessus, on peut voir 2 *règles* qui représentent des implications de la logique et 3 formules atomiques qui sont soit des *faits*, soit des *questions*. On revoit dans ce tableau que les variables quantifiées de la logique perdent leur symbole de quantification en Prolog.

Voici le même tableau en λProlog :

TABLEAU 24. λProlog : correspondance avec la logique

Logique	λProlog
chat(felix)	`chat felix.`
$(\forall x)(ronronne(x) \Leftarrow chat(x))$	`ronronne X :- chat X.`
$(\exists x)ronronne(x)$	`:- ronronne X.`
pere(felix,noireau)	`pere felix noireau.`
$(\forall x)(est-mere(x) \Leftarrow (\exists y)mere(x,y))$	`est-mere X :- mere X Y.`

7 Arithmétique Récursivité Itération

7.1 Arithmétique et récursivité dans les langages fonctionnels

Voici une définition de la fonction factorielle $N! = \prod_{i=1}^{n} i = 1 \times 2 \times 3 \times \ldots \times n$. Elle peut être définie de manière récursive comme suit :

$$fac(n) = \begin{cases} 1 & si(n = 0) \\ n \times fac(n-1) & si(n > 1) \end{cases}$$

FIGURE 29. Définition récursive de factorielle

Cette fonction se définit en CAML comme suit :

```
let rec fact = function
   | 1 -> 0
   | n -> n*(fact (n-1));;
```

7.1.1 Réduction d'une expression

La séquence de réductions correspondant à l'évaluation de l'expression `fact(3)` est ainsi la suivante :

```
fact 3
3 * fact (3 - 1)
3 * fact 2
3 * (2 * fact (2 - 1))
3 * (2 * fact 1)
3 * (2 * (1 * fact (1 - 1)))
3 * (2 * (1 * fact 0))
3 * (2 * (1 * 1))
3 * (2 * 1)
3 * 2
6
```

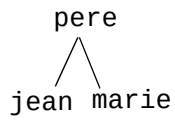
La complexité de cette fonction en temps et en mémoire utilisée est *Ò(n)* pour le calcul de *fact(n)*.

7.2 Arithmétique et récursivité dans les langages relationnels

7.2.1 Arithmétique dans les langages relationnels, où on apprend que $1 + 1 \neq 2$

En programmation logique la construction *1+1* est un terme au même titre que *pere(jean,marie)*. Dans ces deux termes les symboles + et *pere* sont des *constructeurs binaires* (*d'arité 2)*. Les deux termes représentent respectivement des arbres :

TABLEAU 25. Deux arbres construits avec des constructeurs d'arité 2

1+1	
pere(jean,marie)	

Les termes construits en PROLOG ne sont jamais évalués. La seule opération qu'ils subissent est l'unification. L'arbre `1 + 1`, que l'on peut aussi écrire de manière préfixe `((+) 1 1)` n'est pas unifiable avec l'atome 2.

Pour effectuer des évaluations d'expression arithmétiques λPROLOG fournit des opérateurs d'évaluation dont les deux principaux sont `is` et `fis`. Il s'agit de prédicats binaires prédéfinis ayant les types respectifs suivants :

```
type (is) int -> int -> o.
type (fis) float -> float -> o.
```

Comme les connecteurs logiques binaires que nous avons déjà vus `(:-)`, `(,)` et `(;)`, ces deux opérateurs d'évaluation sont des opérateurs binaires infixes. La partie droite doit être un terme construit sans variable au moment de l'exécution du `is` ou du `fis`. Ce terme représente une expression qui est évaluée lorsque le but est choisit par l'interpréteur. Voici les constructeurs d'expressions arithmétiques :

```
type (+, -, *, //, mod, >, <, =<, >=) int -> int -> int.
type ($+, $-, $*, $/, $<, $>, $=<, $>=) float -> float -> float.
```

La partie gauche des deux évaluateurs `is` et `fis` doit être une variable ou une constante de type respectif `int` ou `float`, elle est unifiée avec le résultat de l'évaluation de l'expression qui se trouve à droite. Pour effectuer une addition telle que `Y+Z`, nous devons écrire `X is Y+Z` ou `X fis Y+Z` selon que les variables `X`, `Y` et `Z` prennent des valeurs entières ou décimales.

7.2.2 Fonction factorielle récursive.

Considérons la fonction factorielle $N! = \prod_{i=1}^{n} i = 1 \times 2 \times 3 \times \ldots \times n$.

Sa spécification fonctionnelle récursive est : $fac(n) = \begin{cases} 1 \Leftarrow (n = 0) \\ n \times fac(n-1) \Leftarrow (n \neq 0) \end{cases}$.

On en déduit facilement le prédicat λPROLOG `(fac +N ?R)` qui réussit lorsque $R = N!$:

```
type fac int -> int -> o.
fac 0 1.          % N=0 (1)
fac N R :-        % N≠0 (2)
   N =\= 0,
   N1 is N-1,
   fac N1 R1,
   R is R1*N.
```

Remarquez le but `N =\= 0` qui signifie $N \neq 0$ dans la clause (2). Le type de ce comparateur est :

```
type (=\=) int -> int -> o.
```

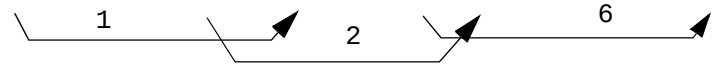
Il existe le même pour les décimaux :

```
type ($=\=) float -> float -> o.
```

Le but `N =\= 0` permet de spécifier que factorielle est une fonction et rend exclusif les deux cas `fac(0, R)` et `fac(N, R)`.

Voici l'exécution du but `:- (fac 3 X)` par l'interpréteur λPROLOG :

```
fac 3 R
   ↓ (2)  σ = {N1 ← 3,R1 ← X}
3 =\= 0, N1_1 is 3-1, fac N1_1 R1_1, X is R1_1*3
   ↓ ...
fac 2 R1_1, X is R1_1*3
   ↓ (2) σ = {N2 ← 2,R2 ← R1_1}
fac 1 R1_2, R1_1 is R1_2*2, X is R1_1*3
   ↓ (2) σ = {N3 ← 1,R3 ← R1_2}
fac 0 R1_3, R1_2 is R1_3*1, R1_1 is R1_2*2, X is R1_1*3
   ↓ (1) σ = {R1_3 ← 1}
R1_2 is 1*1, R1_1 is R1_2*2, X is R1_1*3
```




```
  ...
```

Pour calculer le but `:- (fac n X)` on construit une conjonction de `n` buts de la forme :

```
R1_i is R1_i-1*(n-i+1).
```

La complexité en temps et en mémoire pour le calcul de `(fac +N -R)` est *Ò(N)* comme pour le programme récursif en OCAML donné ci-dessus.

7.2.3 Fonction factorielle itérative - la technique des accumulateurs

Voici un calcul de $n!$ par itération :

```
int fact(int n) {
   int k = 0;
   int product = 1;             // initialization
   while (k < n) {              // test
      k = k + 1;
      product = product * k;    // loop body
   }
   return product;
}
```

La complexité en mémoire de cet algorithme est meilleur que la version précédente récursive, elle est *Ò(Cte)*. La gestion de mémoire est effectuée grâce aux deux affectations :

```
   k = k + 1;
   product = product * k;    // loop body
```

En PROLOG on ne dispose pas de possibilité de réutiliser la mémoire mais dispose de paramètres. Le prédicat λPROLOG `fact` itératif s'écrit comme suit :

```
type fact int -> int -> o.
type fact1 int -> int -> int -> o.

fact N R:-
   fact1 N 1 R.

fact1 0 A A.      % condition de terminaison K = 0

fact1 K A R:-     % invariant de la boucle (∀k(0 < k < n + 1))A_k = ∏_{i=n}^{k} i

   K =\= 0,
   A1 is A*K,
   K1 is K-1,
   fact1 K1 A1 R.
```

Exercice 9.

Écrire l'arbre de recherche correspondant à la question `fact 4 R` *pour la version itérative du prédicat* `fact`. *En raisonnant sur cet arbre de recherche donnez la complexité en temps et en mémoire de ce prédicat.*

Cette technique très connue d'optimisation transforme un programme récursif en un programme itératif. Elle s'appelle la technique de *l'accumulateur.* La technique de l'accumulateur n'est pas spécifique de la programmation logique. Elle peut être employée dans n'importe quel autre langage.

Exercice 10.

Voici la définition récursive de la fonction exponentielle :

$$\exp(i,n) = \begin{cases} 1 \Leftarrow (n = 0) \\ i \times \exp(i, n-1) \Leftarrow (n \neq 0) \end{cases}$$

Écrire et typer le prédicat λPROLOG récursif calculant cette fonction. Sa spécification est `(exp +I +N ?R)` *réussit avec* $R = I^N$.

Exercice 11.

Donnez une version itérative du prédicat `exp` *défini ci-dessus.*

8 Fermeture transitive d'une relation: prédicats récursifs

Considérons une relation binaire qui définit un graphe sans boucle. En programmation il est classique de construire la relation *fermeture transitive* d'un tel graphe grâce à une simple récursivité. Voici un exemple avec la relation *parent*.

Considérons la théorie suivante $\left\{\begin{array}{l}\text{parent(claudie,juliette)}\\ \text{parent(jeanne,claudie)}\end{array}\right\}$. Le prédicat *ancetre* peut être définit comme suit :

$$(\forall x)(\forall y)(\text{ancetre}(x,y) \Leftarrow (\text{parent}(x,y) \vee (\exists z)(\text{parent}(x,z) \wedge \text{ancetre}(z,y))))$$

Voici cette théorie en λPROLOG :

```
kind personne type.
type (claudie, jeanne, juliette) personne.

type (parent, ancetre) personne -> personne -> o.

parent claudie juliette.                    // (1)
parent jeanne claudie.                      // (2)

ancetre(X,Y) :- parent(X,Y).                // (3)
ancetre(X,Y) :- parent(X,Z), ancetre(Z,Y). // (4)
```

Voici l'effacement du but `:- ancetre jeanne juliette.` par l'interpréteur λPROLOG dans le programme `P` ci-dessus :

`P :- ancetre jeanne juliette. //----------> point de choix [1]`
choix de la clause (3)
↓ $\sigma = \{X_1 \leftarrow \text{jeanne}, Y_1 \leftarrow \text{juliette}\}$
`P :- parent jeanne juliette.`
↓
échec

Reprise au point de choix [1]
`P :- ancetre jeanne juliette. //----------> choix [1] terminé`
choix de la clause (4)
↓ $\sigma = \{X_1 \leftarrow \text{jeanne}, Y_1 \leftarrow \text{juliette}\}$
`P :- parent jeanne` Z_1`, ancetre` Z_1 `juliette.`
choix de la clause (2)
↓ $\sigma = \{Z_1 \leftarrow \text{claudie}\}$
`P :- ancetre claudie juliette. //----------> point de choix [2]`
choix de la clause (3)
↓ $\sigma = \{X_2 \leftarrow \text{claudie}, Y_2 \leftarrow \text{juliette}\}$

`P :- parent claudie juliette.`
choix de la clause (1)
↓ $\sigma = \varnothing$
succès

Reprise au point de choix [2]
`P :- ancetre claudie juliette. //----------->` choix [2] terminé
choix de la clause (4)
↓ $\sigma = \{X_2 \leftarrow \text{claudie}, Y_2 \leftarrow \text{juliette}\}$
`P :- parent claudie` Z_2`, ancetre` Z_2 `juliette.`
choix de la clause (1)
↓ $\sigma = \{X_2 \leftarrow \text{claudie}, Y_2 \leftarrow \text{juliette}\}$
`P :- ancetre juliette juliette. //----------->` point de choix [3]
choix de la clause (3)
↓ $\sigma = \{X_3 \leftarrow \text{juliette}, Y_3 \leftarrow \text{juliette}\}$
`P :- parent juliette juliette.`
↓
échec

Reprise au point de choix [3]
`P :- ancetre juliette juliette. //----------->` choix [3] terminé
choix de la clause (4)
↓ $\sigma = \{X_3 \leftarrow \text{juliette}, Y_3 \leftarrow \text{juliette}\}$
`P :- parent juliette` Z_3`, ancetre` Z_3 `juliette.`
↓
échec

Exercice 12.

Effectuer l'effacement (éventuellement prévoyez sans le faire l'effacement) du même but dans le programme `P'` *construit avec les mêmes clauses que celles de* `P` *mais en inversant les deux clauses* `(3)` *et* `(4)`*. Faites vos conclusions.*

Exercice 13.

Effectuer l'effacement du même but dans le programme `P''` *construit avec les mêmes clauses que celles de* `P` *mais en remplaçant la clause* `(4)` *par la clause* `(4')` *suivante (équivalente logiquement)*

```
ancetre(X,Y) :- ancetre(Z,Y), parent(X,Z). // (4').
```

Que se passe t-il ?

9 La coupure

La coupure (ou coût), noté ! (prononcez «cut») en PROLOG, est un prédicat sans paramètre qui n'a aucune sémantique logique si ce n'est qu'il est toujours prouvé avec succès. Sa sémantique est opératoire uniquement. Il sert à restreindre l'arbre de recherche en empêchant le développement de branches qui conduisent à des échecs ou à des solutions que l'on ne veut pas connaître.

On peut comprendre la coupure avec deux visions. Une vision dynamique sur l'arbre de recherche et une vision statique sur le programme.

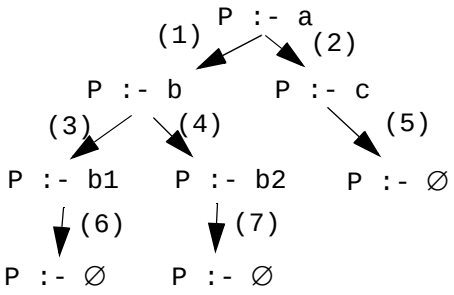
9.1 Vision dynamique de la coupure

Du point de vue de l'arbre de recherche, la preuve du prédicat ! a pour effet de bord de couper des branches en attente de développement. Elle coupe l'ensemble des branches en attente créées depuis l'appel de la clause qui a introduit la coupure.

Voici un programme `P` qui possède plusieurs démonstrations du but `:- a` :

```
type (a,b,c,b1,b2) o.

a :- b.    /* (1) */
a :- c.    /* (2) */

b :- b1.   /* (3) */
b :- b2.   /* (4) */

c.         /* (5) */
b1.        /* (6) */
b2.        /* (7) */
```

Voici l'arbre de recherche du but `P :- a` :

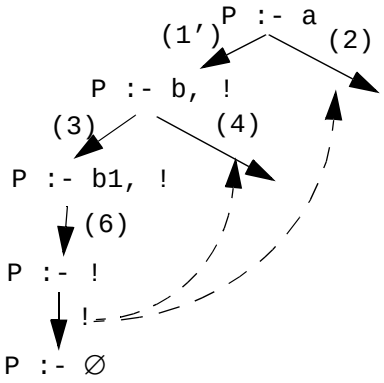


Les trois démonstrations utilisent respectivement les séquences de clauses {1,3,6}, {1,4,7} et {2,5}.

Si on remplace la clause (1) par la clause :

```
a :- b, !.    /* (1') */
```

L'arbre devient :

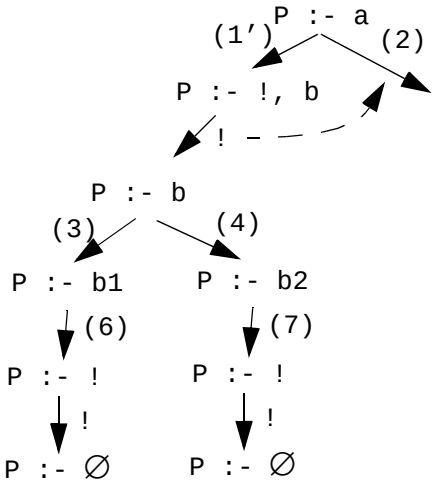


Une seule démonstration est effectuée {1,3,6}.

Si on remplace la clause (1) par la clause :

```
a :- !, b.    /* (1'') */
```

L'arbre devient :



Deux démonstrations sont effectuées {1,3,6}, {1,4,7}.

9.2 Vision statique de la coupure

On interprète la coupure de manière statique en regardant le prédicat dans lequel elle apparaît. Voici un exemple de prédicat avec coupure :

```
A :- B.
A :- B1, B2, ! ,B3, B4.
A :- C.
```

Dans ce prédicat la coupure apparaît dans la seconde clause en position de troisième but avec des buts à droite (`B3, B4`) et des buts à gauche (`B1, B2`).

L'effet d'une coupure telle que ci-dessus ne se comprend à la vue du programme que si on connaît les deux stratégies de choix de PROLOG :

- stratégie $\vee$: les clauses sont choisies en séquence de haut en bas.

- stratégie $\wedge$: les buts sont choisis en séquence de gauche à droite.

La coupure a alors deux effets :

1. Sur les clauses du prédicat où elle apparaît : Les clauses qui suivent celle où apparaît le ! sont ignorées si la coupure vient à être exécutée. Les clauses qui précèdent continuent à jouer leur rôle normalement.
2. Sur les buts en partie gauche de la clause où la coupure se trouve : les buts à gauche de la coupure perdent leur non déterminisme dès que la coupure vient à être exécutée (c'est le cas des buts `B1` et `B2` de l'exemple). Les buts à droite de la coupure continuent à jouer leur rôle normalement (c'est le cas des buts `B3` et `B4` de l'exemple).

10 Les types en λProlog

10.1 Type énumérés

Voici un rappel de quelques types énumérés que nous avons déjà vus :

```
kind personne type.
type (marie, pierre) personne.

kind cardinal type.
type (nord, sud, est, ouest) cardinal.

kind adjectif type.
type (boreal, austral, oriental, occidental) adjectif.
```

10.2 Type simple construit : le jeux de carte

Voici une modélisation du jeux de cartes en λPROLOG :

```
kind (carte, couleur) type.

type (pique, coeur, carreau, trefle) couleur.

type (ass, roi, dame, valet) couleur -> carte.
type petiteCarte int -> couleur -> carte.
```

On définit ici le type `couleur` et ses quatre constantes comme des valeurs énumérée. Pour modéliser les cartes on utilise quatre constructeur unaire qui construisent une carte à partir d'une couleur et un constructeur binaire qui construit une carte à partir d'un entier et d'une couleur. Voici des exemples de cartes :

```
(ass coeur), (valet pique), (petiteCarte 5 carreau) ...
```

10.3 Type simple construit récursif : les entiers de Peano

G. PEANO (XIXe siècle) est l'auteur d'un modèle des entiers naturels $\mathbf{N} = \{0,1,2,3,\ldots\}$. Ce modèle permet de définir tous les nombres entiers naturels à

partir du premier d'entre eux 0 et d'une fonction *successeur* qui, à un entier *n*, associe l'entier *n+1*.

Dans ce qui suit nous allons construire les entiers de PEANO en modélisant directement leur axiomatique. Sachant que la principale propriété des entiers de Peano (ce pourquoi PEANO les a conçu) est que l'on peut construire sur eux une arithmétique qui modélise l'arithmétique des entiers naturels, les programmes que nous allons construire de manière déclarative modéliseront l'arithmétique des entiers naturels.

Vous découvrirez ainsi qu'en PROLOG, et plus encore en λPROLOG, spécifier c'est programmer.

10.3.1 Type «*peano*»

Ici on définit récursivement le type «*peano*».

Pour commencer on se donne deux symboles `{zero,suc}`.

- Une constante `zero` pour représenter l'élément neutre pour l'addition 0 ;
- La fonction unaire `suc(n)` pour représenter la fonction *successeur* introduite par PEANO.

Les deux axiomes suivants spécifient alors récursivement le type `peano` :

$$\left\{\begin{array}{c} \text{zero} \in \text{peano} \\ (\forall n)((n \in \text{peano}) \Rightarrow (\text{suc}(n) \in \text{peano})) \end{array}\right\}$$

FIGURE 30. Spécification récursive des entiers de Peano

De cette spécification on déduit très naturellement la représentation suivante des entiers de peano en λPROLOG :

```
kind peano type.

type zero peano.
type suc peano -> peano.
```

Ce nouveau type récursif permet de représenter l'infinité des entiers naturels à l'aide des deux seuls symboles `{zero/0, suc/1}`. Voici quelques entiers naturels représentés par cette méthode :

TABLEAU 26. Quelques entiers naturels représentés par des termes de type `nat` en λProlog

0	1	2	3	n

TABLEAU 26. Quelques entiers naturels représentés par des termes de type `nat` en λProlog

Terme	*zero*	*(suc zero)*	*(suc (suc zero))*	*(suc (suc (suc zero)))*	*(suc^n zero)*
Arbre	zero	suc | zero	suc | suc | zero	suc | suc | suc | zero	suc | n{ ... | suc | zero

10.3.2 Arithmétique de Peano

A partir de ce type, nous allons construire l'*arithmétique de peano.* Généralement on prends la signature suivante `{0,suc, + , × }` et on définit alors l'arithmétique par 8 axiomes (voir `http://fr.wikipedia.org/wiki/Axiomes_de_Peano`). Les trois premiers axiomes mettent en place les entiers de Peano:

TABLEAU 27. Axiomes définissant les entiers de Peano

	Axiomes	**Signification intuitive**
(1)	$(\forall x)\neg(suc(x) = zero)$	`zero` est le successeur de personne.
(2)	$(\forall x)(\exists y)(\neg(x = zero) \Rightarrow (suc(y) = x))$	Si `x` n'est pas `zero` alors `x` est le successeur de quelqu'un.
(3)	$\{(\forall x)(\forall y)((suc(x) = suc(y)) \Rightarrow (x = y))\}$	Quel que soient `x` et `y`, si le successeur de `x,` et le successeur de `y` sont égaux, alors x et y sont égaux.

Nous ne le démontrerons pas ici, mais il est aisé de se convaincre, en regardant le manuel de référence de λPROLOG, que l'implémentation de l'unification et celle des prédicats prédéfinis d'égalité == et de non égalité `\==`, de type `A -> A -> o` implémentent collaborativement, par construction les trois premiers axiomes de l'arithmétique de PEANO pour le type défini par :

```
kind peano type.

type zero peano.
type suc peano -> peano.
```

Les quatre axiomes suivants implémentent l'addition et le produit. Nous allons les reprendre comme spécification des prédicats `add` et `mul`. Voici ces quatre axiomes :

TABLEAU 28. Axiomes pour les opérations d'additions et de multiplication

	Axiomes	**Signification intuitive**
(4)	$(\forall x)(0 + x = x)$	0 est l'élément neutre pour l'addition
(5)	$(\forall x)(\forall y)(x + suc(y) = suc(x + y))$	$x + (y + 1) = (x + y) + 1$ associativité de l'addition
(6)	$(\forall x)(0 \times x = 0)$	Quelque soit x le produit de 0 et de x est toujours 0
(7)	$(\forall x)(\forall y)(x \times suc(y) = (x \times y) + x)$	Définition récursive de la multiplication comme une suite d'additions

Le dernier axiomes de l'arithmétique de PEANO est une propriété qui autorise les raisonnements par récurrence. Il s'agit plutôt d'un schéma d'axiomes, le voici :

TABLEAU 29. Schéma d'axiome exprimant le raisonnement par récurrence

	Axiomes	**Signification intuitive**
(8)	$P(0) \wedge (\forall x)(P(x) \Rightarrow P(x+1)) \Rightarrow P(y)$	Pour toute propriété P (formule logique de premier ordre avec des variables libres dont x), si on a P(0) et si pour tout x $P(x) \Rightarrow P(x+1)$ alors *P(y)* pour tout y.

Voici l'addition en λPROLOG selon les axiomes (4) et (5) :

```
type add peano -> peano -> peano -> o.

add zero N N.              /* (4) */(∀x)(0 + x = x)
add (suc N) M (suc R) :-   /* (5) */(∀x)(∀y)(x + suc(y) = suc(x + y))
   add N M R.
```

Remarque : Dans ce qui suit, pour des raisons de concision, nous représentons le terme `(suc (suc (suc (suc zero))))` par $(suc^4\ zero)$.

Considérez la question «*trouver R tel que* $R = 3 + 5$». Elle se représente dans le programme P par le but $:-\ (add\ (suc^3\ zero)\ (suc^5\ zero)\ R)$.

Nous donnons ci-dessous l'arbre de rechercher correspondant à cette question dans le programme P. Dans la première étape de cette recherche, l'interpréteur λPROLOG choisit la clause (5) du programme P car la tête de cette clause s'unifie avec le but. Nous montrons ci-dessous cette tête de clause instanciée et le but.

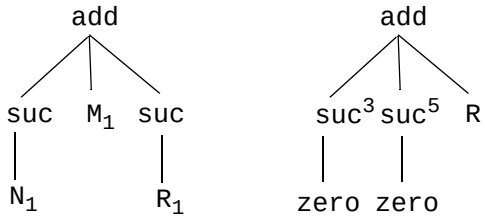


FIGURE 31. Deux termes à unifier

Exercice 14.

Unifier ces deux arbres et trouver une substitution plus générale (un pgu) qui unifie ces deux termes.

L'arbre de recherche qui suit correspond au calcul par λPROLOG de $R = 3 + 5$ par le programme P.

```
P :- add (suc^3 zero) (suc^5 zero) R.
   ↓ (5) {N1 <- (suc^2 zero), M1 <- (suc^5 zero), R <- (suc R1)}
P :- add (suc^2 zero) (suc^5 zero) R1.
   ↓ (5) {N2 <- (suc zero), M2 <- (suc^5 zero), R1 <- (suc R2)}
P :- add (suc zero) (suc^5 zero) R2.
```

```
   ↓ (5) {N3 <- zero, M3 <- (suc^5 zero), R2 <- (suc R3)}
P :- add zero (suc^5 zero) R3.
   ↓ (4) {N4 <- (suc^5 zero), R3 <- N4}
P :- ∅
```

La recherche se termine par un but vide, c'est à dire par un succès. Il suffit alors de parcourir l'arbre et de considérer les substitutions accumulées pendant la recherche pour trouver la valeur du résultat. Voici les substitutions cumulées intéressante :

$$\{R \leftarrow suc(R_1), R_1 \leftarrow suc(R_2), R_2 \leftarrow suc(R_3), R_3 \leftarrow N_4, N_4 \leftarrow suc^5(zero)\}$$

On en réduit que `R=(suc^8 zero).`

Exercice 15.

Que signifie la question `:- add X (suc^5 zero) (suc^8 zero)` *?*

Construisez l'arbre de recherche correspondant à cette question.

Exercice 16.

Que signifie la question `:- add X Y (suc^3 zero)` *?*

Construisez l'arbre de recherche correspondant à cette question.

Exercice 17.

Donnez les modes possibles pour le prédicat `add` *et ceux qui ne le sont pas. Expliquez.*

Constatez que la définition fonctionnelle que nous avons utilisé pour concevoir notre prédicat `add` *est une vision extrêmement limitative de ce prédicat.*

Il arrive souvent en PROLOG que l'on conçoive délibérément une prédicat (un programme) avec une vision de départ de type fonctionnel (vision plus simple à appréhender) et que l'on se convainque ultérieurement que le programme possède des propriétés héritées de la logique qui lui donnent d'autres modes de fonctionnement. Nous retrouvons ce phénomène pour les manipulations de *listes, d'ensembles, d'arbres* ou tout simplement de *relations* (voir le prédicat `roseDesVents` défini en introduction de ce cours).

Vous verrez en TP l'implémentation complète de l'arithmétique de PEANO.

10.4 Les listes : type récursif paramètré

Les listes sont des structures de données offertes par tous les langages de programmation. λPROLOG ne déroge pas à la règle.

En λPROLOG les listes sont des suites homogènes de valeurs parenthésées par des crochets «[» «]» et séparées par des virgules «,», par exemple la liste `[1,2,3]` est la liste des trois premiers entiers. `1,2,3` sont les éléments de la liste, la taille de cette liste est 3. Par contre la liste à 4 éléments `[1,jean,suc(zero),5.77]` qui est une liste correcte en PROLOG est une liste incorrecte en λPROLOG car ses éléments ne sont pas de type homogène. La liste vide se note `[]`.

10.4.1 Spécification de la liste

En fait la notation [1,2,3] est un sucre syntaxique pour les objets construits selon la spécification récursive suivante :

1. l'atome () est la liste vide
2. Une liste non vide se compose de deux parties: une partie `tete` (premier élément) et une `queue` (la liste des éléments restants).

En reprenant l'idée ci-dessus d'un constructeur à deux places pour représenter les liste non-vides, on définit souvent formellement une liste monomorphe dont les éléments sont de type `E` par le type abstrait `L` suivant :

```
nil: () → L
cons: E × L → L
first: L → E
rest: L → L
```

avec les axiomes pour tout e ∈ E et pour tout l ∈ L $first(cons(e,l)) = e$ et $rest(cons(e,l)) = l$

Les propriétés suivantes pour tout *e,ei* ∈ E tout *l, li* ∈ L sont alors toujours vérifiées : $cons(e,l) \neq l$, $cons(e,l) \neq e$ et $cons(e_1,l_1) = cons(e_2,l_2)$ si $(e_1 = e_2) \wedge (l_1 = l_2)$

Cette spécification définit un type «liste de E» à partir d'un type «E».

10.4.2 Polymorphisme générique en λPROLOG

Les types de λPROLOG sont des types du premiers ordre avec des variables (voir plus loin). Ils permettent d'introduire la généricité au sens de ML (voir R. Milner. A theory of type polymorphism in programming. J. Computer and System Sciences, 17:348–375, 1978.). On introduit d'abord le constructeur de type `list` dépendant d'un type comme suit :

```
kind list type -> type.
```

Le constructeur de type `list` peut être vu comme une fonction des types dans les types, d'où la flèche dans sa déclaration. Il s'agit d'un constructeur (d'une fonction) d'arité 1. Il est alors possible de déclarer des constructeurs de termes génériques en utilisant des variables de type comme suit

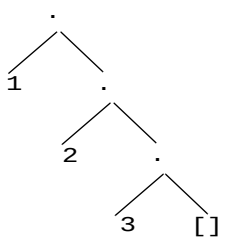
```
type [] (list A).                    /* pour la liste vide */
type '.' A -> (list A) -> (list A). /* pour la liste non vide */
```

Les déclarations du constructeur 0 aires `[]` et du constructeur binaire `'.'` doivent être comprises comme si elles étaient quantifiées par ∀A devant la déclaration. On dit ici que `[]` est une liste «de quelque chose.» qui est vide. On dit aussi que les constructeurs `[]` et `'.'` sont *génériques.*

La liste notée dans les programmes `[1,2,3]` est en fait représentée en interne par le terme :

```
('.' 1 ('.' 2 ('.' 3 [])))
```

C'est à dire par l'arbre :



Dans un terme `('.' A L)` `A` est appelé la *tête* et `L` la *queue*. On retrouve les fonctions `first` et `rest` utilisées pour la définition du type abstrait `list` en Section 10.4.1, page 52

Dans une instance de liste les constructeurs prennent un type qui est une instance du type générique de leur définition où la variable A est remplacée par une instance de type. Par exemple dans les termes suivants `('.' 1 [])` et `('.' true [])` les deux constructeurs ont respectivement les types :

```
[]  : (list int).
'.' : int -> (list int) -> (list int).
```

dans le terme `('.' 1 [])` (i.e. dans la liste singleton `[1]`) et

```
[]  : (list o).
'.' : o -> (list o) -> (list o).
```

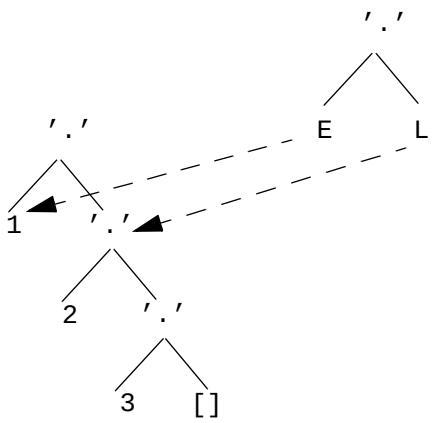
dans le terme `('.' true [])` (i.e. dans la liste singleton `[true]`)

C'est pour cette raison que l'on dit que ces constructeurs sont *génériques*. Chaque occurrence d'utilisation dans un programme possède son propre type qui peut être différentes de toutes les autres occurrences du programme.

10.4.3 Accès aux deux champs d'une liste en λPROLOG

Dans la définition du type abstrait `list` que nous avons donné en Section 10.4.1, page 52 deux fonctions sont importantes, les fonctions `first` et `rest` elles donnent respectivement accès à la tête d'une liste et à la queue de la liste. PROLOG et λPROLOG n'ont nul besoin de définir de telles fonctions car l'unification par son principe même peut donner accès aux deux champs de n'importe quelle liste.

Il suffit d'unifier la liste donnée, par exemple la liste `[1,2,3]`, avec le terme général suivant `('.' E L)`. La substitution qui unifie ces deux termes est la suivante : `{E← 1, L← [2,3]}`



Ce principe est tellement utilisé en programmation logique que les langages de programmation logique ont été munis d'un sucre syntaxique pour parler du terme `('.' E L)`. Il s'écrit avec une barre comme suit : `[E|L]`.

Remarque importante à propos des types de `E` et `L` dans un terme `[E|L]`. Si le terme `[E|L]` est de type `(list A)` alors `E` est de type `A` et `L` est de type `(list A)`.

Le sucre syntaxique décrit ci-dessus a été généralisé de sorte que la barre peut être mise en dernière place de toute liste, par exemple :

`[1,2,3|L]` signifie `('.' 1 ('.' 2 ('.' 3 L)))`.

Unifier `[1,2,3]` avec `[1,2,3|L]` produit la substitution résultat `{L← []}`

Voici de multiples notations équivalentes de la liste `[1,2,3]` :

```
[1,2,3|[]]
[1,2|[3]]
[1,2|[3|[]]]
[1|[2,3]]
[1|[2,3|[]]]
...
```

10.4.4 Programmation sur les listes

Les listes induisent une programmation récursive naturelle où le cas d'arrêt est souvent, mais pas nécessairement, la liste vide `[]`. Nous allons voir cela maintenant.

Concaténation de listes

Le profil du prédicat `conc` est le suivant `(conc +L1 +L2 -L)` où `L` est le résultat de la concaténation des listes `L1` et `L2`.

Le type du prédicat conc est générique, il peut concaténer des listes de n'importe quoi, pourvu que les deux listes possèdent la même instance de type. Sa déclaration est la suivante :

```
type conc (list A) -> (list A) -> (list A) -> o.
```

La généricité du prédicat conc apparaît ici grâce à la présence de la variable A dans la définition du type. La seule contrainte qui est imposée ici sur l'utilisation de conc est que les trois listes (les deux listes à concaténer et le résultat de la concaténation) sont des listes d'éléments de même type.

Voici une spécification fonctionnelle récursive de la concaténation conc :

$$conc(l_1,l_2) = \begin{cases} l_2 \Leftarrow (l_1 = [\ ]) \\ [a|conc(l,l_2)] \Leftarrow (l_1 = [a|l]) \end{cases}$$

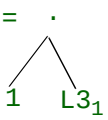
On déduit naturellement de cette spécification le prédicat suivant (à deux cas) :

```
conc [] X X.
conc [A|L1] L2 [A|L3] :-
   conc L1 L2 L3.
```

Voici une question : `:- (conc [1,2] [3,4] R).`

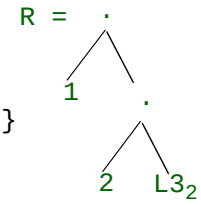
Voici l'arbre de recherche pour cette question :

```
P :- conc [1,2] [3,4] R.
```



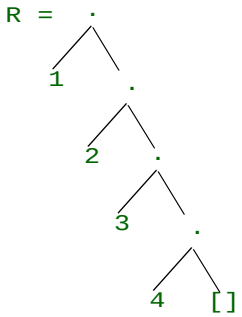
↓ {A_1 <- 1, $L1_1$ <- [2], $L2_1$ <- [3,4], R <- [A_1|$L3_1$}

```
P :- conc [2] [3,4] L3₁.
```



↓ {A_2 <- 2, $L1_2$ <- [], $L2_2$ <- [3,4], $L3_1$ <- [A_2|$L3_2$}

```
P :- conc [] [3,4] L3₂.
```



↓ {X_1 <- [3,4], $L3_2$ <- X_1}

```
P :- ∅
```

Complexité du prédicat conc

On voit sur cet arbre de recherche que le temps d'exécution du prédicat `conc` pour une exécution dans le mode `(conc + + -)` est proportionnel à la taille `N` de la première liste. Celle-ci doit être recopiée élément par élément avant d'être en mesure d'accrocher la seconde liste à la fin de la liste recopiée. La seconde liste n'est pas parcourue, ni touchée, une référence sur elle est simplement passée d'appel en appel telle quelle jusqu'à la fin de la récursivité. Ceci donne une complexité en temps d'exécution de l'ordre de *Ò(n)* si n est la taille de la première liste.

Pour la taille mémoire requise pour une exécution dans le mode `(conc + + -)`, on constate sur l'arbre que la complexité est de l'ordre d'une constante *Ò(cts).*

Exercice 18.

Le prédicat `conc` *peut être utilisé pour décomposer une liste en sous-listes. Donner son mode de fonctionnement dans ce cas. Écrire la question qui décompose la liste* `[1,2,3]`. *Faire l'arbre de recherche correspondant à cette question.*

Y a-t-il d'autres modes de fonctionnement acceptables pour ce prédicat ?

Exercice 19.

Pour construire une liste `R` *dont la tête est l'élément* `E` *et la queue est une une liste* `L` *nous avons vu qu'il suffit d'écrire* `[E|L]`.

Comment mettre un élément `E` *en queue d'une liste* `L` *pour donner une liste* `R` *?*

10.4.5 Schéma de programme inductif sur la structure des listes

On remarque dans la définition du prédicat `conc` (Section 10.4.4, page 54) que, de part le fonctionnement même de l'unification, les deux cas définis par les deux clauses du prédicat sont exclusifs :

- Le cas de la liste vide (arrêt de la récursivité) ;
- Le cas de la liste non vide (cas récursif).

De nombreux prédicats suivent ce schéma de programme. Les exercices qui suivent en sont des exemples.

Exercice 20.

Voici la spécification de la fonction longueur d'une liste :

$$\text{long}(l) = \begin{cases} 0 \Leftarrow l = \text{nil} \\ \text{long}(r) + 1 \Leftarrow (l = a \bullet r) \end{cases}$$

En suivant cette spécification écrire et typer le prédicat `(length +L -N)` *tel que* `N` *est la longueur de la liste* `L`. *Écrire l'arbre de recherche pour la question* `:- length [1,2,3] N`.

Quelle est la complexité en temps et en mémoire de votre prédicat ?

Pensez vous que vous pouvez améliorer les choses ? Si oui donnez une nouvelle version du prédicat. Expliquez.

Exercice 21.

Écrire et typer le prédicat générique `(applatir +L -R)` *tel que* `L` *est liste de listes de n'importe quoi et* `R` *est le résultat de la concaténation de toutes les listes de* `L`.

10.4.6 Prédicat `member`

10.4.7 Prédicat `reverse`

10.4.8 Programmation récursive avec accumulateur

10.5 Les arbres

10.5.1 Arbres binaires

On veut construire des arbres binaires dont chaque noeud binaire est décoré d'un terme d'un certain type et chaque feuille de l'arbre supporte un terme d'un type éventuellement différent de celui des termes aux nœuds. Il s'agit donc d'un type générique paramètré par deux types. On se donne deux constructeurs de termes, un pour les nœuds, un pour les feuilles. On définit alors le type et ses deux constructeurs comme suit :

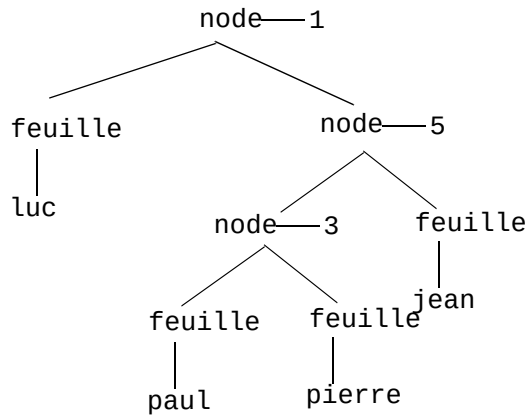
```
kind arbreBin type -> type -> type.

type node A -> (arbreBin A B) -> (arbreBin A B) -> (arbreBin A B).
type feuille B -> (arbreBin _ B).
```

Considérez que vous disposez du type et des atomes suivants :

```
kind personne type.

type (luc,paul,pierre,jean) personne.
```

Voici un exemple d'arbre du type `(arbreBin int personne)` :

```
(node 1
  (feuille luc)
  (node 5
    (node 3
      (feuille paul)
      (feuille pierre)
    )
    (feuille jean)
  )
)
```



10.5.2 Compter les feuilles d'un arbre binaire

Voici la spécification de la fonction générique *nbFeuille(a)* qui compte le nombre de feuilles dans l'arbre *a* :

$$\text{nbFeuille(a)} = \begin{cases} 1 \Leftarrow a = \text{feuille}(\ldots) \\ \text{nbFeuille}(a_g) + \text{nbFeuille}(a_d) \Leftarrow (a = \text{node}(\ldots, a_g, a_d)) \end{cases}$$

Voici une réalisation en λPROLOG de cette spécification :

```
type nbFeuille (arbreBin A B) -> int -> o.

nbFeuille (feuille _) 1.
nbFeuille (node _ Ag Ad) N :-
   nbFeuille Ag Ng,
   nbFeuille Ad Nd,
   N is Ng+nd.
```

10.5.3 Lister les feuilles d'un arbre binaire

Exercice 22.

Écrire et typer le prédicat générique `(listFeuille +A -L)` *qui construit la liste* `L` *des feuilles d'un arbre binaire* `A` *donné en entrée tel que ci-dessus.*

On appel «chemin vers une feuille `F` depuis la racine» dans un arbre tel que ci-dessus, la liste ordonnée des valeurs rencontrées aux nœuds pour arriver directement à la feuille `F` depuis la racine. Par exemple le chemin pour aller à la feuille `(feuille pierre)` dans l'arbre ci-dessus est `[1,5,3]`.

10.5.4 Lister non déterministement les chemins vers les feuilles

Exercice 23.

Trouvez une méthode de représentation en λPROLOG *des paires suivantes :*

<chemin vers une feuille `F` depuis la racine, `F`>

Déclarez un type générique et un constructeur générique de terme permettant de représenter de telles paires.

Écrire et typer le prédicat `(Chemin +A -P)` *qui génère par non déterminisme individuellement pour un arbre donné* `A`*, tel que ci-dessus, toutes les paires* `P` *suivantes :*

<chemin vers une feuille F depuis la racine,`F`>.

10.6 Les arbres à branches multiples

Un arbre à branches multiple est composé d'une racine et d'un ensemble éventuellement vide de descendants qui sont eux mêmes des arbres à branches multiples. Les nœuds d'un arbre à branches multiples sont décorés d'une valeur. Un arbre à branches multiples n'est jamais vide, il y a toujours une racine,

éventuellement sans descendants. Voici une proposition de représentation des arbres à branches multiples :

```
kind multiwayTree type -> type.
```

`multiwayTree` est un type générique qui dépend d'un autre type, celui des informations décorant les nœuds. On se donne un unique constructeur de termes :

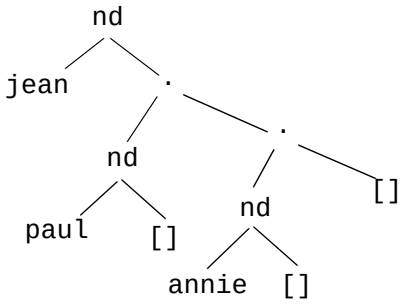
```
type nd A -> (list (multiwayTree A)) -> (multiwayTree A).
```

Considérons un domaine de personnes défini comme suit :

```
kind personne type.

type (jean,paul,annie) personne.
```

Voici alors un arbre qui pourrait être arbre généalogique :

```
(nd jean [nd paul [], nd annie []])
```



10.6.1 Compter les nœuds d'un arbre à branches multiples

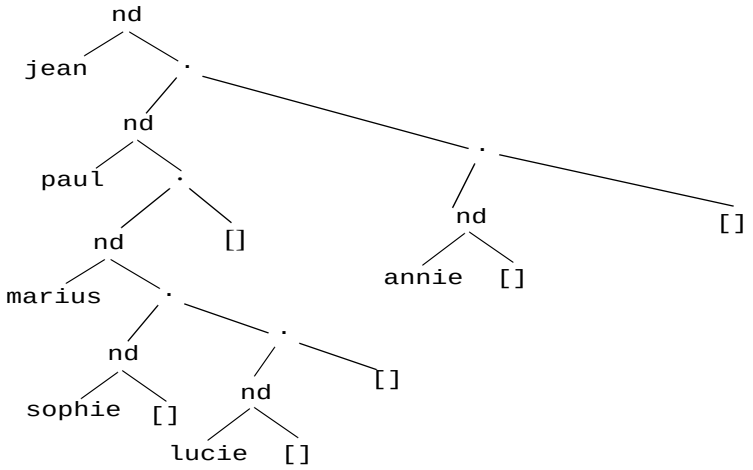
Voici le prédicat `(nbNodes +A -N)` tel que `N` est le nombre de nœuds de `A`

```
type nbNodes (multiwayTree A) -> int -> o.
type nbNodes1 (list (multiwayTree A)) -> int -> o.

nbNodes (nd _ L) R :-
   nbNodes1 L R1,
   R is R1+1.

nbNodes1 [] 0.
nbNodes1 [A|L] R :-
   nbNodes A R1,
   nbNodes1 L R2,
   R is R1+R2.
```

10.6.2 Liste des nœuds par un parcours de bas en haut de droite à gauche

Le problème consiste à constuire la séquence des nœuds rencontés par un parcours de bas en haut et de gauche à droite. Par exemple pour l'arbre :

```
(nd jean [nd paul [nd marius [nd sophie [], nd lucie []]], nd annie
     []])
```



La liste des nœuds correspondant à un parcours de bas en haut et de gauche à droite est : `L=[sophie, lucie, marius, paul, annie, jean]`

Le prédicat `(bottomUpLeftRight +A -L)` tel que `L` est la liste des nœuds rencontrés par un parcours de bas en haut et de gauche à droite de l'arbre `A` est le suivant :

```
type bottomUpLeftRight (multiwayTree A) -> (list A) -> o.
type bottomUpLeftRight1 (multiwayTree A) -> (list A) -> (list A) -
     > o.
type bottomUpLeftRight2 (list (multiwayTree A)) -> (list A) ->
     (list A) -> o.

bottomUpLeftRight1 (nd Nd L) A R :-
   bottomUpLeftRight2 L [Nd|A] R.

bottomUpLeftRight2 [] A A.
bottomUpLeftRight2 [E|L] A R :-
   bottomUpLeftRight2 L A R1,
   bottomUpLeftRight1 E R1 R.

bottomUpLeftRight A L :-
   bottomUpLeftRight1 A [] L.
```

Remarquez que pour éviter d'utiliser la concaténation qui aurait donné un prédicat avec une complexité en temps de $Ò(n^2)$ si n est le nombre de nœuds de l'arbre, nous avons utilisé deux prédicats intermédiaires avec accumulateurs (voir le problème de l'inversion naïve ...). La complexité en temps de la version ainsi obtenue est $Ò(n)$.

10.7 Les structures de données incomplètes

10.7.1 Les listes en différence

10.7.2 Les arbres incomplets

11 λProlog et les λtermes

11.1 Définition

Les termes (ou expressions) de λPROLOG s'écrivent avec :

- les caractères «λ», «(», «)»
- des λvariables (V = x; y; z:::),
- des constantes (C= `jean, pierre, rouge, pere`, ...),
- des variables logiques ou inconnues (U=X,Y,N1,...).

Définition : l'ensemble des λtermes simplement typés de λPROLOG est le plus petit ensemble vérifiant les propriétés suivantes :

- λvariable $V_t \subset \Lambda_t$ «*toute λvariable de type t est un λterme de type t*» ;
- variable logique (inconnue) $U_t \subset \Lambda_t$ «*toute variable logique de type t est un λterme de type t*»
- constante $C_t \subset \Lambda_t$ «*toute constante de type t est un λterme de type t*»
- abstraction $\forall x(x \in V_{t1})$, $\forall A(A \in \Lambda_t)$ alors $\lambda x \cdot A \in \Lambda_{t1 \to t}$ «*si x est une λvariable de type t1, a est un λterme de type t alors <u>lambda x a</u> est un λterme de type* $t1 \to t$» ;
- application $\forall A(A \in \Lambda_{t1 \to t}), \forall B(B \in \lambda_{t1})$ alors $(A\ B) \in \Lambda_t$ «*si a est un λterme de type* $t1 \to t$, *b est un λterme de type t1 alors <u>a appliqué à b</u> est un λterme de type t*»

Pour l'écriture l'opération d'application est prioritaire devant l'abstraction.

Exemples : x, $jean$, $x\ y$, $\lambda x \cdot x$, $(f\ x)\ y$, $\lambda x \cdot x\ x$, $\lambda x \cdot x\ x$, $\lambda x \cdot pere\ jean\ x$, ...

Intuitivement, $\lambda x \cdot y$ est la fonction qui à X «*paramètre*» associe Y «*modèle de terme*» ; F X construit un nouveau terme par application de F «*fonction*» à X «*argument*».

11.2 Curryfication

On peut tout faire avec des fonctions à un argument.

Comparer $mul(x, y) = x \times y$ et $m(x) = mul_x(y) = x \times y$ leurs types respectifs sont :

- *mul* : $(x, y) \to x * y$

- m : x $\rightarrow$ (y $\rightarrow$ x * y)
- mul_x : (y) $\rightarrow$ x * y par exemple : mul_2(y) $\rightarrow$2 * y

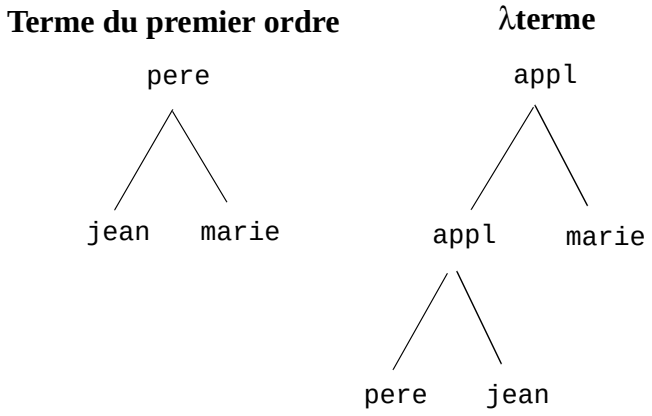
et on a $mul(2, 3) = (m(2))(3) = mul_2(3)$.

Notation :

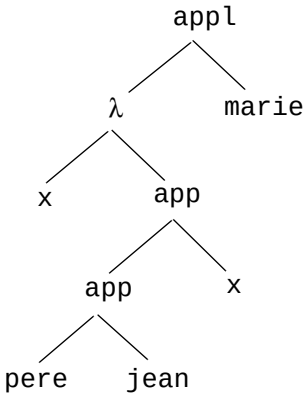
- les parenthèses à gauche sont inutiles, `((A B) C)` est identique à `(A B C)` ;
- un seul λ pour des abstractions successives, $\lambda x\, y\, z.\, A$ pour $\lambda x\,.\,\lambda y\,.\,\lambda z\,.\,A$.

11.3 Structure arborescente des λtermes

Voici deux représentations d'une application `(pere jean marie)` :



Voici la représentation arborescente du λterme (λx · (pere jean x) marie) :



11.4 Réprésentation des polynômes d'entiers à une indéterminée

11.4.1 Représentaion en PROLOG

En Prolog on représente classiquement les polynômes à une indéterminée par poly(X,X+1), poly(X,(X+3)*(X-2)).

1. eval

2. dérivation

11.4.2 Représentation en lProlog

1. eval
2. dérivation

11.4.3 Normalisation

On considère les polynômes d'entiers que l'on représente par des listes d'entiers en commençant par les coefficients de plus petit degré. Ces polynômes sont normalisés : le coefficient de plus grand degré doit être non nul.

1. normalisation
2.
3. addition
4. division euclidienne

12 Prédicats d'ordre supérieur

Vous allez voir ici les fonctions d'ordre supérieur et les fonctions annonymes.

12.1 Prédicats d'ordre supérieur pourquoi ?

Grâce à sa notation curryfiée, à ses λtermes et à son unification sur les λtermes simplement typés qui apporte la βréduction, λPROLOG manipule les prédicats comme des objets de premier ordre. Ceci signifie qu'ils peuvent être stoqués dans des termes, passés en paramètre et rendus en résultat puis ultérieurement utilisés en tant que prédicat.

Nous allons introduire l'utilisation des prédicats d'ordre suppérieur avec un exemple simple. Voici le prédicat `(double +I -R)` tel que `R=I+I` et le prédicat `(carre +I -R)` tel que `R=I*I` :

```
type (double,carre) int -> int -> o.

double I R :- R is I+I.
carre I R :- R is I*I.
```

Voici maintenant les prédicats `(fois4 +I -R)` tels que `R=(I+I)+(I+I)` et `(puissance4 +I -R)` tel que `R=(I*I)*(I*I)` :

```
type (fois4,puissance4) int -> int -> o.

fois4 I R :- double I R1, double R1 R.
puissance4 I R :- carre I R1, carre R1 R.
```

On remarque que ces deux prédicats suivent le même schéma de programme :

```
:- P I R1, P R1 R.
```

On en déduit tout naturellement le prédicat d'ordre supérieur `(appliquer2fois +P +I -R)` tel que $R = F(F(I))$. Il se programme comme suit en λPROLOG :

```
type appliquer2fois (A -> A -> o) -> A -> A -> o.

appliquer2fois P I R :- P I R1, P R1 R.
```

Remarquez tout d'abord la généricité de ce prédicat sur laquelle nous avons été vigilant. `appliquer2fois` ne s'applique pas seulement aux prédicats de type `(int -> int -> o)`, il est générique et s'applique à tout prédicat ayant le schéma de type `(A -> A -> o)`.

Remarquez ensuite que le premier paramètre de `appliquer2fois` est un prédicat, pour cette raison on dit que `appliquer2fois` est d'ordre supérieur.

Les fonctions `fois4` et `puissance4` deviennent alors :

```
fois4 I R :- appliquer2fois double I R.
puissance4 I R :- appliquer2fois carre I R.
```

Ici non seulement la similarité entre les deux prédicats devient évidente mais on peut imaginer que si quelqu'un veut améliorer les performances des prédicats `fois4` et `puissance4` il lui suffit de changer l'implémentation de `appliquer2fois`. Tous les prédicats définis à l'aide de `appliquer2fois` bénéficieront de l'amélioration.

Voici la résolution du but `:- appliquer2fois double 4 R :`

```
:- appliquer2fois double 4 R.
↓   {P ← double, R1 ← R}
:- double 4 R1, double R1 R.
↓   {R1 ← 8}
:- double 8 R.
↓   {R ← 16}
∅
```

À l'aide du prédicat `double` et des arguments `4` et $R1_1$ le but `(double 4` $R1_1$`)` de type `o` est construit dynamiquement. Dés qu'il se trouve en position de prédicat devant l'interpréteur λPROLOG il est βréduit puis interprété. Le même procédé se retrouve pour le second but construit dynamiquement. PROLOG qui n'a pas cette possibilité de manipuler des prédicats comme objets de première classe, ni d'effectuer des βréductions, a introduit le prédicat évaluable `=..` (univ) comme succédanné d'ordre supérieur. Cela reste cependant un artifice qui possède des limites.

12.2 Prédicats anonymes

Il est possible de définir un prédicat anomyme par un λterme de λPROLOG. Voici un exemple d'appel : `:- appliquer2fois x\ z\ (z is x+x) 4 R`. Ceci est équivalent à l'appel : `appliquer2fois double 4 R`.

Il serait aussi possible d'écrire la question suivante :

```
:- P = (appliquer2fois x\ z\ (z is x+x)), P 4 R1, P 5 R2.
```

On obtiendrait un succès avec la subtitution `{R1 ← 16, R2 ← 20}`. Dans la question ci-dessus, la valeur de `P` est de type `(int -> int -> o)`, elle résulte de l'application de `appliquer2fois` à un argument `x\ z\ (z is x+x)`. Cette valeur est un prédicat qui est ensuite appliqué à ses deux arguments. C'est la notation curriée de λPROLOG et la βréduction provoquée par l'unification des λtermes simplement typés qui nous permettent une telle écriture naturelle et sans artifice.

Exercice 24.

Écrire et typer le prédicat `(compose +P1 +P2 +A -R)` *tel que* `R=P1(P2(A))`.

Exercice 25.

Écrire et typer le prédicat `(iter +N +P +A -R)` *tel que* $\left(\overbrace{\mathrm{P(P(\ldots(P(A))))}}^{\mathrm{Nfois}}\right) = \mathrm{R}$. *De manière informelle,* `P` *est appliqué* `N` *fois. D'abord* `P` *est appliqué à* `A`*, puis* `P` *est appliqué au résultat* `P(A)`*, etc. Le résultat final est placé dans* `R`.

Exercice 26.

Donnez les résultats (succès ou échec) des deux questions suivantes :

```
nFois 10 x\ y\ (y is 2+x) 0 N.
nFois 10 x\ y\ (y is 2*x) 1 N.
```

En cas de succès donnez la ou les substitutions résultantes.

12.3 Abstraire les itérations sur les listes

Grâce à l'ordre supérieur on peut abstraire les parcours sur les listes.

12.3.1 Appliquer une fonction à tous les éléments d'une liste

Voici le prédicat `(doubleAll +L -R)` tel que `R` est la liste des entiers doubles des entiers trouvées dans la liste `L`. `R` et `L` ont la même taille

```
type doubleAll (list int) -> (list int) -> o.

doubleAll [] [].
doubleAll [I|L] [J|R] :-
   J is 2*I,
```

```
   doubleAll L R.
```

On trouve dans ce prédicat un des schémas de programme récursif les plus employé sur les listes. On peut en déduire le prédicat `(mapPred +P +L -R)` qui prend en paramètre un prédicat `P`, une liste `L` et applique `P` à tous les éléments de `L`, les résultats étant déposés dans `R`. Voici ce prédicat en λPROLOG :

```
type mapPred (A -> B -> o) -> (list A) -> (list B) -> o.

mapPred _ [] [].
mapPred P [A|L] [B|R] :-
   P A B,
   mappred L R.
```

`mapPred` est dit d'ordre supérieur car son premier paramètre est un prédicat.

Remarquez aussi que comme pour le prédicat `appliquer2fois` défini précédemment, nous avons préservé au maximum la généricité du prédicat `mapPred`. Si le prédicat `P` passé en paramètre transforme des valeurs de type `A` en des valeurs de type `B`, le prédicat `(mapPred P)` transforme des listes de `A` en des listes de `B` :

$$\tau(P) = A \to B \to o$$
$$\tau(\text{mapPred}(P)) = \text{list}(A) \to \text{list}(B) \to o$$

Soit le prédicat `(double +I -R)` tel que `R=2*I` défini en Section 12.3.1, page 65. On peut réécrire le prédicat `doubleAll` à l'aide du prédicat `mappred` et du prédicat `double` comme suit :

```
type doubleAll (list int) -> (list int) -> o.

doubleAll L R :- mapPred double L R.
```

En utilisant une fonction anonyme on peut éviter de déclarer le prédicat `double`. Voici la nouvelle proposition

```
doubleAll L R :- mapPred x\ y\ (y is 2*x) L R.
```

Voici par exemple le prédicat `(i2fAll +L -R)` qui transforme une liste `L` d'entiers en la liste `R` des réels correspondants aux entiers de `L`, en utilisant `mapPred` et la fonction `i2f` de la librairie «`arith`» de λPROLOG :

```
type i2fAll (list int) -> (list float) -> o.

i2fAll L R :- mapPred i2f L R.
```

Le prédicat `mapPred` que nous venons d'étudier est tellement utile qu'il a été mis dans la librairie standard «`lists`» de λPROLOG sous le nom `maplist`. Voir `maplist` dans la documentation de λPROLOG.

12.3.2 Plier à droite

Considérons les fonctions `P(x,y) = z` de type `(`T_1 `->` T_2 `->` T_2`)`, et les constantes `C` de type T_2. Pour tout couple `<P,C>` on peut définir la fonction suivante :

$$\text{plierDroite}_{P,C}(L) = \begin{cases} C \Leftarrow L = nil \\ P(A,\text{plierDroite}_{P,C}(R)) \Leftarrow L = [A|R] \end{cases}$$

Voici le prédicat `(plierDroite +P +C +L -R)` qui implémente cette fonction avec le résultat $R = \text{plierDroite}_{P,C}(L)$:

```
type plierDroite (A -> B -> B -> o) -> B -> (list A) -> B -> o.

plierDroite P Z [] Z.
plierDroite P Z [A|L] R :-
   plierDroite P Z L R1,
   P A R1 R.
```

Grâce à cet itérateur, le traitement uniforme d'une liste consite à choisir un prédicat `P` et une constante `C`. Voici des exemples.

TABLEAU 30.

Description	Prédicat	Constante
Somme des éléments d'une liste	`x\ y\ z\(z is x+y)`	`0`
Compter les éléments d'une liste	`x\ y\ z\(z is y+1)`	`0`
Produit des éléments d'une liste	`x\ y\ z\(z is x*y)`	`1`
Concaténer une liste de listes	`append`	`[]`
Inversion naïve d'une liste	`e\ l\ r\(append l [a] r)`	`[]`

Exercice 27.

Typer et dites ce que fait le prédicat suivant :

```
plierDroite e\ l\ r\ (r=[e|l])
```

Exercice 28.

En utilisant seulement le prédicat `plierDroite` *écrire et typer le prédicat* `(listAll +P +L -R)` *tel que* `R` *est la liste de tous les éléments de* `L` *qui vérifient le prédicat* `P`*. et seulement ceux là*