Ne vous moquez pas de l'oiseau moqueur : un aperçu de la logique combinatoire

Yannis Haralambous

Ne vous moquez pas de l'oiseau moqueur : un aperçu de la logique combinatoire

Avec des applications en linguistique mathématique

par Yannis Haralambous*

Résumé.

Nous proposons une découverte d'un domaine de la logique qui aura prochainement 100 ans : la logique combinatoire, dans le cadre des fonctions primitives récursives. Notre point de départ est le conte mathématique « Ne vous moquez pas de l'oiseau moqueur » de Raymond Smullyan. Après une brève description de la manière dont les objets de la logique combinatoire représentent les fonctions primitives récursives, nous donnons deux applications de celle-ci : l'algèbre de Boole et l'analyse syntaxique du langage naturel.

I Introduction

Une certaine forêt enchantée est habitée d'oiseaux qui parlent : c'est ainsi que commence le conte mathématique *Ne vous moquez pas de l'oiseau moqueur* [16] de Raymond Smullyan. Le célèbre auteur de recueils de casse-têtes — à travers lesquels il vulgarise un certain nombre de résultats importants comme le théorème d'incomplétude de Gödel — est mort récemment à l'âge respectable (et symbolique [1]) de 97 ans. L'*oiseau moqueur* a été traduit, par ordre chronologique, en espagnol [17], en allemand [19], en slovène [18] et en hongrois [20], mais malheureusement pas encore en français.

Il s'agit d'une introduction à une discipline appelée « logique combinatoire ». Nous nous proposons dans cet article de découvrir cette discipline à partir du texte de Smullyan, dans le cadre des fonctions primitives récursives. Puis nous donnerons quelques exemples d'applications, aussi bien en mathématiques qu'en linguistique. Pour commencer, parlons de *fonctions primitives récursives*.

* Département Informatique, IMT Atlantique - UMR CNRS 6285 Lab-STICC, yannis.haralambous@imt-atlantique.fr

1. Il s'agit du plus grand nombre premier inférieur à 100.

II Les fonctions primitives récursives

Le lecteur [2] connaît la notion de *fonction* : il s'agit d'associer à chaque élément d'un certain *domaine de définition* une, et une seule, valeur dans un *domaine de valeurs*. Ainsi la fonction $f(x)=\cos(x)$ associe à chaque x (domaine de définition : l'ensemble $\mathbb{R}$) une valeur dans l'ensemble $[-1,1]$; la fonction $f(x,y)=\sqrt{x^2+y^2}$ associe à chaque vecteur de $\mathbb{R}^2$ son module, et ainsi de suite.

Nous allons nous intéresser à un ensemble particulier de fonctions, celles qui sont *calculables par une procédure effective*, autrement dit : *dont un programme d'ordinateur peut calculer les valeurs exactes*. On se restreint à celles dont le domaine de définition est un sous-ensemble de $\mathbb{N}^k$ et qui sont à valeurs dans $\mathbb{N}$.

Définition 1. *Une* ***fonction primitive récursive de base*** *peut être de trois types :*

1. *la fonction constante* 0*;*
2. *la fonction* successeur σ*;*
3. *une fonction de* projection π^i_k.

2. Dans cet article « le lecteur » est synonyme de l'inclusif « le·a lecteur·rice ».

La fonction *successeur* envoie tout entier naturel n à $n+1$. La fonction de projection π_k^i envoie un k-uplet d'entiers vers le i-ème entier (en particulier, π_1^1 est l'identité, $\pi_2^1(x,y)=x$ et $\pi_2^2(x,y)=y$ pour tous x,y).

Définition 2. *Si g est une fonction $\mathbb{N}^\ell \to \mathbb{N}$ et $h_1,\ldots,h_\ell$ des fonctions $\mathbb{N}^k \to \mathbb{N}$, alors la **composition** de g et des $h_1,\ldots,h_\ell$ est la fonction $f:\mathbb{N}^k\to\mathbb{N}$ définie par*

$$f(n_1,\ldots,n_k) := g(h_1(n_1,\ldots,n_k),\ldots,h_\ell(n_1,\ldots,n_k)).$$

*Si g est une fonction $\mathbb{N}^k\to\mathbb{N}$ et h est une fonction $\mathbb{N}^{k+2}\to\mathbb{N}$, alors la fonction $f:\mathbb{N}^{k+1}\to\mathbb{N}$ est **définie à partir de g et h par récursion primitive**, si et seulement si :*

$$f(n_1,\ldots,n_k,0) = g(n_1,\ldots,n_k),$$
$$f(n_1,\ldots,n_k,\sigma(m)) = h(n_1,\ldots,n_k,m,f(n_1,\ldots,n_k,m)).$$

où σ est la fonction successeur.

La première définition correspond à la notion ordinaire de composition de fonctions. Pour comprendre la deuxième définition, prenons un exemple : la *fonction somme* qui à (n,m) associe $n+m$: elle est définie par (a) le fait que $n+0=n$ pour tout n, et (b) le fait que le successeur de la somme est égal au premier terme plus le successeur du deuxième, donc $k=1$, somme$(n_1,0)=n_1$ (g est l'identité) et somme$(n_1,\sigma(m))=\sigma(\text{somme}(n_1,m))$, donc $h=\sigma\circ\pi_3^3$. De la même manière on peut définir une fonction *produit* en écrivant produit$(n_1,0)=0$ et produit$(n_1,\sigma(m))=\text{somme}(n_1,\text{produit}(n_1,m))$, autrement dit : $k=1$, $g=0$ et $h=\text{somme}\circ(\pi_3^1,\pi_3^3)$.

Définition 3 ([23, 6.2.1])**.** *Une **fonction primitive récursive** est :*

1. *une fonction primitive récursive de base, ou*
2. *une fonction obtenue à partir d'un nombre quelconque de compositions et de récursions primitives appliquées aux fonctions primitives récursives de base.*

La fonction $f(n)=3n+2$ est donc une fonction primitive récursive, puisqu'elle peut s'écrire

$$f(n)=\text{somme}(\text{produit}(\sigma(\sigma(\sigma(0)))),n),\sigma(\sigma(0))).$$

Dans la suite, quand nous parlerons de « fonction » il sera entendu « fonction primitive récursive ». Nous verrons à la section VI que les objets de la logique combinatoire, appelés *combinateurs*, représentent ce type de fonctions : à chaque fonction on peut associer un combinateur, qui devient un « programme » si on considère la logique combinatoire en tant que langage de programmation.

III La logique combinatoire

Dans la suite nous adopterons les conventions usuelles de la logique combinatoire, qui peuvent paraître déroutantes au premier abord. En effet, on va partir d'un ensemble quelconque, et on va généraliser l'*application de fonction* à une opération appelée « application » et qui sera définie *pour tout couple d'éléments de l'ensemble* : si A et B sont deux éléments de l'ensemble, on aura aussi bien l'application (AB) que (BA), ainsi que (AA) et (BB). Si A est une fonction f et B un élément x, alors (AB) sera bien l'application ordinaire $f(x)$ — et les notations (BA), (AA) et (BB) seront contraires à l'intuition. C'est sans doute cela qui a poussé un des fondateurs de la théorie, Robert Feys, à écrire en 1946 [6, p. 75] :

> reconnaissons qu'aucune partie de la logique formalisée ne présente un premier abord aussi insolite et aussi rebutant [que la logique combinatoire]. [Elle] ne dérange pas seulement nos habitudes d'écriture ; elle nous jette hors de toutes nos habitudes de pensée.

L'utilisation répétée de l'opération d'application nous fournit ce que nous appellerons des *termes*. Plus rigoureusement, voici comment on définit un *terme* par induction (un procédé qui généralise la récurrence [14, chap. 2]) :

Définition 4. *Soit E un ensemble contenant des constantes et des variables. Un **terme** sur E est défini par induction :*

- *si c est une constante de E, alors c est un terme ;*
- *si x est une variable sur E, alors x est un terme ;*
- *si M et N sont des termes alors (MN) est un terme (on dira qu'on* applique M à N *et que (MN) est l'*application *de M à N).*

Comme on le voit, ce ne sont pas seulement les constantes et les variables qui peuvent s'appliquer les unes aux autres, mais aussi les termes, quelque soit leur complexité. On obtient donc (MN), $((MN)N)$ (c'est-à-dire (MN) appliqué à N), $(((MN)N)(MN))$ (c'est-à-dire $((MN)N)$ appliqué à (MN)), et ainsi de suite...

Le lecteur aura remarqué que les expressions du style $(((MN)N)(MN))$ contiennent un nombre démesuré de parenthèses, auquel on n'est pas habitué dans les mathématiques de base. Cela est dû à l'absence d'une propriété importante : l'*associativité*. Que ce soit en analyse, en géométrie ou en algèbre, les opérations que l'on voit au cours de l'enseignement secondaire et du 1er cycle supérieur sont, dans leur écrasante majorité, associatives : que l'on écrive $((a+b)+c)+d$ ou $(a+b)+(c+d)$ (où « $+$ » est l'addition de nombres naturels), le résultat est le même. Ce n'est pas le cas ici : dans l'univers des termes, $((ab)c)$ est l'application

de (ab) à c, alors que $(a(bc))$ est l'application de a à (bc), les termes $((ab)c)$ et $(a(bc))$ sont différents.

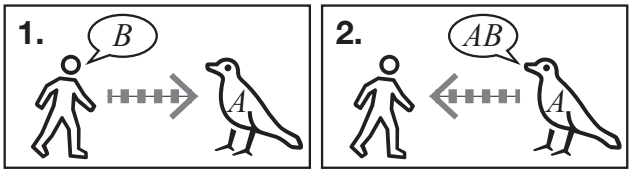
Nous allons adopter une règle qui réduira, autant que possible, le nombre de parenthèses : on décide que, par défaut, *la priorité sera à gauche et qu'on ne mettra de parenthèse que dans le cas contraire*. Ainsi, on peut écrire $fxyz$, qui signifie $(((fx)y)z)$. Si c'est $((fx)(yz))$ que l'on souhaite, on écrira $fx(yz)$. De même, on écrira MN sans parenthèses pour (MN).

Attention : il ne faut pas confondre le terme xyz avec le mot xyz du langage formel d'alphabet $\{x,y,z\}$ (cf. [11, défi. 2]). Dans un mot de langage formel, les symboles sont simplement *concaténés*, comme les lettres d'un mot de langage naturel. Par contre, quand on écrit le terme xyz il est entendu qu'il y a une interaction entre les sous-termes : « x s'applique à y, et le terme (xy) qui en résulte s'applique à z ».

Il est temps maintenant de laisser Smullyan nous entraîner dans le monde des termes de la logique combinatoire, qui pour lui seront... des oiseaux ! (Dans la section qui suit, la scène se joue entièrement dans la forêt enchantée de Smullyan, nous avons inséré quelques commentaires sur les aspects mathématiques de cette forêt et de ses habitants, en caractère bâton.)

IV La forêt enchantée

La forêt enchantée de Smullyan est peuplée d'oiseaux qui parlent et avec qui on peut interagir. Hélas, leur vocabulaire se limite à des noms d'oiseaux. On procède donc comme dans la figure ci-dessous :



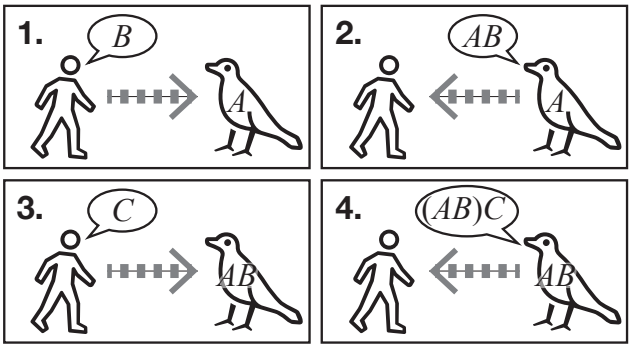
où l'on s'adresse à un oiseau qui s'appelle « A » en prononçant le nom « B » et l'oiseau en question répond par « (AB) », l'*application de A à B* comme expliqué dans la section précédente (et que l'on note aussi « AB » sans parenthèses, si on adopte la notation simplifiée).

On peut considérer qu'un oiseau A est une fonction f_A de l'ensemble des noms d'oiseaux vers lui-même : on lui fournit une valeur B et elle répond par $f_A(B)$.

Notons que la manière par laquelle un oiseau réagit à la totalité des noms avec lequels il peut être appellé le définit entièrement. On est dans une « forêt extensionnelle », cela signifie que si deux oiseaux réagissent de la même manière à tous les noms dont on peut les appeler, alors il s'agit du même oiseau.

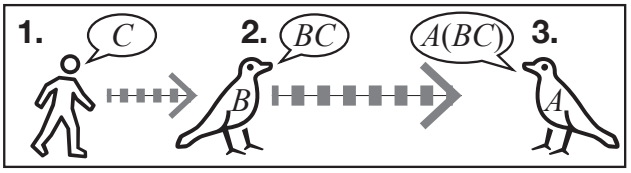
La remarque ci-dessus illustre bien le fait qu'on est dans un monde mathématique : un objet est entièrement défini par ses propriétés. Si une fonction f a les mêmes valeurs que sin dans l'ensemble de définition de celle-ci, alors f *est égale à la fonction* sin. Ici, si $Ax = Bx$ pour tout x, alors on aura $A = B$.

Nous avons déjà mentionné que ABC, c'est-à-dire $(AB)C$ n'est pas la même chose que $A(BC)$. Voyons comment cela se traduit en termes d'oiseaux : le cas $(AB)C$ correspond à la situation où l'humain dit A à l'oiseau B qui répond par AB, et ensuite s'adresse à l'oiseau AB en disant C, et celui-ci répond $(AB)C$:



Mathématiquement, cela correspond à la fonction f_A appliquée à B et ensuite le résultat de cette application, à son tour, appliqué à C. Donc $f_A(B)$ n'est pas un oiseau, mais une *fonction sur l'ensemble des oiseaux*. Grâce à cette astuce, on peut décrire des fonctions de plusieurs variables : si, par exemple, $f : E^2 \to E$ est une fonction à deux variables, on peut définir $f^* : E \to E^E$ par $f^*(x) : y \mapsto f(x,y)$, autrement dit $(f^*(x))(y) = f(x,y)$. On appelle f^* la ***curryfication***[3] de f.

De même, le cas $A(BC)$ est celui de l'humain qui appelle l'oiseau B par le nom C, celui-ci dit BC à l'oiseau A, qui répond $A(BC)$:



En reprenant les mêmes notations que ci-dessous, mathématiquement cet exemple correspond à f_B appliqué à C et ensuite la fonction f_A appliquée à $f_B(C)$, on obtient donc $f_A(f_B(C))$, autrement dit, la *composée* des fonctions f_A et f_B appliquée à C.

Et puisqu'on parle de composition, Smullyan introduit la notion suivante :

Définition 5. *On dit qu'un oiseau M* compose *un oiseau A avec un oiseau B, si pour tout oiseau x on a la condition $Mx = A(Bx)$.*

3. La technique de remplacement d'une fonction de n variables par une suite de n fonctions qui « absorbent progressivement les variables une par une », a été nommée d'après le mathématicien Curry, alors qu'elle a été inventée à l'origine par Schönfinkel. Comme le dit avec beaucoup d'humour [12, p. 3], personne n'aurait envie d'utiliser le terme *schönfinkelisation*...

Autrement dit : quand on s'adresse à M on obtient les mêmes réponses que comme si on s'adressait à B qui parle à A, qui nous répond.

La notion de composition mérite quelques explications. Dans l'univers des fonctions de l'enseignement secondaire ou du premier cycle d'études, on a le grand luxe de *toujours* pouvoir composer des fonctions f et g — à condition bien sûr que l'image de la première soit comprise dans le domaine de définition de la deuxième. Quand cette condition est assurée, $f \circ g$ existe toujours en tant que fonction. Ce n'est pas le cas dans la forêt enchantée : on peut, pour tous A, B et x, écrire $A(Bx)$ (les parenthèses sont ici primordiales), et ce terme fait évidemment partie de l'ensemble. Mais cela ne veut nullement dire qu'il existe un autre élément M tel que $Cx = A(Bx)$ pour tout x, donc la « composée » de A et de B n'existe pas forcément dans un ensemble quelconque E.

C'est la *condition* Ⓒ *de compositionnalité* ci-dessous qui nous garantit l'existence de la composée de deux termes quelconques.

Définition 6. *On dit que dans une forêt on a la* ***condition de compositionnalité*** Ⓒ *si quels que soient les oiseaux A et B il existe toujours un M (également noté* $A \circ B$*) qui les compose.*

Définition 7. *Un* ***oiseau moqueur*** *(*mockingbird *en anglais) est un oiseau* **M** *tel que pour tout oiseau x, on ait* $\mathbf{M}x = xx$.

Autrement dit : un oiseau moqueur se moque de l'humain puisqu'il répète ce qu'aurait dit tout oiseau si on l'avait appelé par son propre nom.

M est un ***répétiteur*** dans le sens où il répète le terme auquel il est appliqué. On peut s'amuser à appliquer **M** à lui-même : **MM** appliqué à un terme x redonne **MM**x, sans le dupliquer ; par contre, $\mathbf{M}(\mathbf{M}x) = (\mathbf{M}x)(\mathbf{M}x) = xx(xx)$. La composition $\mathbf{M} \circ \mathbf{M}$ répète donc quatre fois le terme auquel on l'applique, en suivant le schéma $xx(xx)$, c'est-à-dire xx appliqué à xx.

Dorénavant on va considérer que dans la forêt enchantée il existe un oiseau moqueur **M**, et que la condition Ⓒ est respectée. Rappelons que *puisqu'un oiseau est entièrement et uniquement défini par son comportement*, lorsqu'un oiseau moqueur **M** — ou tout autre type d'oiseau nommé — existe, on considérera qu'il est *unique*.

Définition 8. *On dit qu'un oiseau A* aime *un oiseau B si* $AB = B$.

On retrouve ici la notion de ***point fixe*** : B est un point fixe de l'opération d'application de A.

Proposition 9 (Propriété de l'amour, [16, p. 74])**.** *Dans une forêt enchantée dans laquelle il existe un* **M***, la condition* Ⓒ *entraîne la condition* Ⓐ*, dite « propriété de l'amour » :*

Ⓐ *tout oiseau aime au moins un autre oiseau.*

Autrement dit : toute fonction a un point fixe.

Démonstration. Soit A un oiseau, montrons qu'il existe un oiseau aimé par A. Par Ⓒ on sait qu'on peut composer tout couple d'oiseaux, et l'existence de **M** fait partie de nos hypothèses. Composons A avec **M**, il existe donc C (la composée) tel que $Cx = A(\mathbf{M}x)$ pour tout x. L'astuce est de prendre $x = C$, qui donne $A(\mathbf{M}C) = CC$. Mais **M** est un oiseau moqueur, donc $CC = \mathbf{M}C$, et on obtient $A(CC) = CC$, autrement dit : A aime CC. Cette démonstration est constructive : pour tout oiseau on peut ainsi obtenir un oiseau aimé par lui.

Définition 10. *On appelle un oiseau* **K** *un* ***kamichi*** *(en anglais :* kestrel*) si pour tous x et y,* $\mathbf{K}xy = x$.

K est un ***éliminateur*** puisqu'il supprime un argument parmi ceux qu'on lui fournit, en l'occurrence le deuxième. Il est intéressant d'appliquer **K** à lui-même : **KK** (et tout $\mathbf{K}\cdots\mathbf{K}$ pair) élimine le premier et le troisième arguments fournis : $\mathbf{KK}xyz = \mathbf{K}yz = y$; par contre **KKK** (et tout $\mathbf{K}\cdots\mathbf{K}$ impair) est égal à **K** : $\mathbf{KKK}xy = \mathbf{K}xy = x$. Prenons maintenant les composés de **K** avec lui-même ($\mathbf{K}(\mathbf{K}x)$, $\mathbf{K}(\mathbf{K}(\mathbf{K}x))$, etc.). Si on fait suivre ces termes par autant de termes qu'il y a des **K**, alors ils éliminent tous en le laissant que le premier : $\mathbf{K}(\mathbf{K}x)yz = \mathbf{K}xz = x$, etc.

Si **K** élimine le deuxième terme fourni, le lecteur se demande sans doute comment éliminer le *premier* terme. Pour cela il suffit de prendre **KI**, où **I** est l'identité (cf. ci-dessous). En effet $\mathbf{KI}xy = \mathbf{I}y = y$.

Définition 11. *On appelle* ***identité*** **I** *un oiseau tel que* $\mathbf{I}x = x$ *pour tout x.*

Comme le souligne Smullyan, on peut, à première vue, considérer que l'oiseau identité est stupide, puisqu'il répond par le nom par lequel on l'appelle, quelque soit ce nom. Mais il y a une autre manière de voir **I** : c'est l'oiseau le plus altruiste et passionné de la forêt puisqu'il *aime tout le monde* !

Définition 12. *Un* ***loriot***[4] **L** *(en anglais :* lark*) est un oiseau tel que* $\mathbf{L}xy = x(yy)$ *pour tous x et y.*

On appelle *compositeur* un terme qui introduit des parenthèses parmi ses arguments, par exemple $\mathbf{B}xyz = x(yz)$ que l'on verra par la suite. **L** est un compositeur mais aussi un répétiteur, puisqu'il y a *et* parenthésage *et* répétition.

Le loriot est très bénéfique à la libido de la forêt :

4. Pour respecter le principe acronymique, nous avons donné des noms non conformes à ceux de Smullyan à cinq oiseaux : le loriot **L** (*lark* = alouette pour Smullyan), le kamichi **K** (*kastrel* = crécerelle), le weka **W** (*warbler* = fauvette), le sterne **S** (*starling* = étourneau), le tarin **T** (*thrush* = grive). Nous avons choisi le loriot pour la lettre **L** en hommage à Yvonne Loriod, épouse d'Olivier Messiaen, qui a créé son *Catalogue d'oiseaux* (œuvre belle et déroutante, dont le deuxième mouvement est justement intitulé *Le loriot*) le 15 avril 1959 à la salle Gaveau.

Proposition 13 ([16, p. 80])**.** *Quand il existe un loriot dans une forêt, on a la propriété de l'amour* Ⓐ*.*

Démonstration. Soit A un oiseau quelconque, montrons qu'il existe B tel que $AB = B$. Comme on sait qu'un loriot **L** existe dans la forêt, on a $\mathbf{L}Ay = A(yy)$ pour tout y. En particulier, si on remplace y par $\mathbf{L}A$, on obtient $(\mathbf{L}A)(\mathbf{L}A) = A((\mathbf{L}A)(\mathbf{L}A))$. Donc A aime $(\mathbf{L}A)(\mathbf{L}A)$ et on a la propriété de l'amour.

Le théorème nous dit que si on a **L** dans l'ensemble, alors tout terme a un point fixe : pour tout A, un point fixe sera $(\mathbf{L}A)(\mathbf{L}A)$.

Les kamichis **K** et les loriots **L** ne s'entendent pas bien :

Proposition 14. *Si un loriot* **L** *et un kamichi* **K** *sont distincts,* **L** *ne peut jamais aimer* **K***. Un kamichi* **K** *peut aimer un loriot* **L***, mais uniquement dans le cas où toute la forêt aime* **L***.*

Démonstration. Montrons d'abord un lemme :

Lemme 15 ([16, p. 78])**.** *Si* **K** *aime* $\mathbf{K}x$*, alors* **K** *aime* x*.*

La démonstration est triviale : si $\mathbf{K}(\mathbf{K}x) = \mathbf{K}x$ alors $\mathbf{K}(\mathbf{K}x)y = \mathbf{K}xy$ pour tout y, donc $\mathbf{K}x = x$, ce qui signifie que **K** aime x. □

Supposons maintenant que **L** aime **K**. Alors $\mathbf{LK} = \mathbf{K}$ et donc aussi $\mathbf{LKK} = \mathbf{KK}$. Mais par définition des loriots, $\mathbf{LKK} = \mathbf{K}(\mathbf{KK})$ donc $\mathbf{KK} = \mathbf{K}(\mathbf{KK})$ et donc $\mathbf{KK}xyzt = \mathbf{K}(\mathbf{KK})xyzt$ pour tous x,y,z et t, et donc $\mathbf{K}yzt = \mathbf{KK}yzt = \mathbf{K}zt = z$, le premier terme étant égal à yt. On a donc $yt = z$ pour tous y,t,z, donc $yt = \mathbf{L}$ mais aussi $yt = \mathbf{K}$, mais alors $\mathbf{K} = \mathbf{L}$ ce qui est impossible car contraire à notre hypothèse.

Inversement, si $\mathbf{KL} = \mathbf{L}$ alors pour tout y on a $\mathbf{KL}y = \mathbf{L}y$ et la partie gauche est, par définition des kamichis, **L**. Donc $\mathbf{L}y = \mathbf{L}$ et, en particulier, on a $\mathbf{LL} = \mathbf{L}$. Prenons maintenant un oiseau quelconque A, alors $A\mathbf{L} = A(\mathbf{LL})$ ce qui, par définition des loriots, est $(\mathbf{L}A)\mathbf{L}$ et donc $A\mathbf{L} = (\mathbf{L}A)\mathbf{L} = \mathbf{L}A\mathbf{L} = \mathbf{KL}A\mathbf{L} = \mathbf{LL} = \mathbf{L}$. Dire que pour tout A on a $A\mathbf{L} = \mathbf{L}$ veut bien dire que tout le monde aime **L**.

Il est temps de nous demander où cette série de découvertes d'oiseaux va nous mener. Le lecteur a sûrement remarqué que pour certains oiseaux nous utilisons des caractères spéciaux ($\mathbf{M},\mathbf{K},\mathbf{I},\mathbf{L},\ldots$). Notre but est de montrer qu'un oiseau *quelconque* de la forêt peut être réécrit comme un terme utilisant uniquement les oiseaux d'une liste très restreinte parmi eux. Nous verrons que certaines combinaisons de ces oiseaux « privilégiés » constituent, dans ce sens, des *bases* de l'ensemble des oiseaux de la forêt.

Mais avant de le faire, continuons notre visite de la forêt : nous nous proposons de découvrir encore quelques oiseaux tout aussi beaux qu'intéressants.

Définition 16. *Un* ***oiseau bleu*** **B** *(en anglais :* bluebird*) est défini par* $\mathbf{B}xyz = x(yz)$ *pour tous* x,y,z*.*

Il s'agit d'un terme qui introduit des parenthèses entre les arguments qu'on lui fournit. En réitérant **B** par composition ou par application, on obtient toutes sortes de parenthésages, en gardant toujours l'ordre des arguments fournis : $\mathbf{BB}xyzw = xy(zw)$, $\mathbf{BBB}xyzw = x(yzw)$ qui est aussi $\mathbf{B}(\mathbf{B}x)yzw = x(yzw)$, etc.

B s'appelle ***compositeur*** pour une très bonne raison : il définit à lui tout seul l'opération de composition. En effet, pour tous x,A et B, on a $\mathbf{B}ABx = A(Bx)$ et donc $\mathbf{B}AB = A \circ B$.

Définition 17. *Un* ***weka*** **W** *(an anglais :* warbler*) est un oiseau tel que* $\mathbf{W}xy = xyy$ *pour tous* x *et* y*.*

Attention : ne pas confondre weka ($\mathbf{W}xy = xyy$) et loriot ($\mathbf{L}xy = x(yy)$) !

W est clairement un répétiteur puisqu'il prend un argument et le duplique. Mais, à la différence de **M**, **W** opère « avec un décalage de 1 » : il laisse le premier argument inchangé et duplique le deuxième. L'itération multiple de **W** est très intéressante : **WW** est un triplicateur : $\mathbf{WW}x = \mathbf{W}xx = xxx$. À partir de trois **W** la réécriture tourne en rond, puisqu'en appliquant le premier **W** au deux autres, on obtient de nouveau **WWW**, ce terme ne peut donc pas être réécrit. Quand on compose **W** n fois avec lui-même on obtient n répétitions du deuxième argument : $\mathbf{W}(\mathbf{W}x)y = \mathbf{W}xyy = xyyy$, $\mathbf{W}(\mathbf{W}(\mathbf{W}x))y = \mathbf{W}(\mathbf{W}x)yy = \mathbf{W}xyyy = xyyyy$, etc.

Le théorème suivant, et sa démonstration, montrent les liens entre weka **W**, kamichi **K**, oiseau moqueur **M** et oiseau identité **I** :

Proposition 18 ([16, p. 99])**.** *Toute forêt contenant un weka* **W** *et un kamichi* **K** *doit contenir un oiseau moqueur* **M***.*

Démonstration. En effet, pour tout x, $\mathbf{WI}x = \mathbf{I}xx = xx$ ce qui est aussi le cas de $\mathbf{M}x = xx$ pour tout x, donc on peut dire que **WI** et **M** ont les mêmes propriétés, et donc que $\mathbf{WI} = \mathbf{M}$, donc **M** existe dans la forêt.

Nous pouvons aussi montrer qu'un weka **W** et un kamichi **K** permettent de montrer l'existence d'un oiseau identité **I**. Encore une fois, il suffit d'appliquer le premier au deuxième : **WK** est un oiseau identité **I** puisque pour tout x, $\mathbf{WK}x = \mathbf{K}xx = x = \mathbf{I}x$.

Définition 19. *Un* ***cardinal*** **C** *est un oiseau tel que* $\mathbf{C}xyz = xzy$ *pour tous* x,y,z*.*

C est un ***permutateur*** puisqu'il permute deux de ses arguments, sans changer leur nombre et sans introduire de parenthèse. Si on applique **C** à lui-même, on obtient une autre permutation : $\mathbf{CC}xyz = yzx$. En l'appliquant deux fois, on obtient une permutation de deux termes, mais pour cela il faut l'appliquer à cinq termes : $\mathbf{CCC}xyzuv = \mathbf{C}x\mathbf{C}yzuv = xy\mathbf{C}zuv = xyzvu$. À partir de **CCCC**, on a le

même comportement que **CCC**, quel que soit le nombre de **C**. Et en ce qui concerne la composition, de manière tout à fait prévisible, quand on compose **C** avec lui-même on obtient l'identité : $\mathbf{C}(\mathbf{C}x)yz = \mathbf{C}xzy = xyz$.

Proposition 20 ([16, p. 100])**.** *Toute forêt contenant un cardinal* **C** *et un kamichi* **K** *doit contenir un oiseau identité* **I**.

Démonstration. Quelque soit A, $\mathbf{CK}A$ est un oiseau identité : $\mathbf{CK}Ax = \mathbf{K}xA = x$, en particulier **CKC** et **CKK** sont des identités.

Définition 21. *Un* ***tarin*** **T** *(en anglais :* thrush*) est un oiseau tel que* $\mathbf{T}xy = yx$ *pour tous* x, y.

T est le cas le plus simple de permutateur. En appliquant **T** à lui-même, on obtient **C** : $\mathbf{TT}xyz = x\mathbf{T}yz = xzy = \mathbf{C}xyz$ pour tous x, y, z. À partir de **TTT**, on obtient systématiquement **TT** quel que soit le nombre de **T**.

Proposition 22 ([16, p. 100])**.** *Un cardinal* **C** *et appliqué à un oiseau identité* **I** *donne un tarin.*

Démonstration. On a $\mathbf{CI}xy = \mathbf{I}yx = yx$ pour tous x, y, et d'autre part, par définition du tarin : $\mathbf{T}xy = yx$ pour tous x, y, donc **CI** a le même comportement qu'un tarin.

Définition 23. *Un* ***rouge-gorge*** **R** *(en anglais :* robin*) est un oiseau tel que* $\mathbf{R}xyz = yzx$ *pour tous* x, y, z.

R est encore un permutateur. En appliquant **R** à lui-même, et en l'appliquant cinq fois (à x, y, z, u, v, quelques soient ces termes), on obtient une permutation qui n'affecte que les trois derniers : $\mathbf{RR}xyzuv = xy\mathbf{R}zuv = xyuvz$. En l'appliquant trois fois, on a un comportement identique à celui de **C** : $\mathbf{RRR}xyz = \mathbf{R}x\mathbf{R}yz = \mathbf{R}yxz = xzy = \mathbf{C}xyz$. À partir de **RRR** quelque soit le nombre de **R** appliqués, son comportement ne change plus.

Proposition 24 ([16, p. 101])**.** *On peut obtenir un rouge-gorge* **R** *à partir d'un oiseau bleu* **B** *et d'un tarin* **T**. *À partir d'un rouge-gorge* **R***, on peut obtenir un cardinal* **C***, et inversement.*

Corollaire 25. *À partir d'un oiseau bleu* **B** *et d'un tarin* **T** *on peut obtenir un cardinal* **C**.

Démonstration. On cherche une combinaison de **B** et de **T** telle que, appliquée à xyz quelconque, elle ait le comportement de **R**, c'est-à-dire $\mathbf{R}xyz = yzx$. **TB** et **BB** n'aboutissent à rien, mais $\mathbf{BT}xyz = \mathbf{T}(xy)z = z(xy)$. On constate donc que l'effet de **B** est de réunir xy en un terme et ensuite d'appliquer **T** à $(xy)z$, autrement dit : **B** réunit deux termes avec un décalage de 1. On peut donc s'attendre à ce qu'une double application de **B** réunisse deux termes avec un décalage de 2 : $\mathbf{BBT}xyz = \mathbf{B}(\mathbf{T}x)yz = (\mathbf{T}x)(yz) = \mathbf{T}x(yz) = (yz)x = yzx$ pour tous x, y, z, qui est le comportement de **R**.

On a vu comment obtenir un cardinal à partir d'un rouge-gorge. Pour aller dans l'autre sens il suffit de prendre **C** deux fois : $\mathbf{CC}xyz = \mathbf{C}yxz = yzx$, pour tous x, y, z.

Définition 26. *Un* ***sterne*** **S** *(en anglais :* starling*) est un oiseau tel que* $\mathbf{S}xyz = xz(yz)$ *pour tous* x, y, z.

S est en même temps répétiteur, permutateur et compositeur. Le sterne et le kamichi sont des oiseaux très importants puisqu'on verra par la suite qu'en les combinant, à eux seuls, on peut obtenir le comportement de *tous les oiseaux*.

Proposition 27 ([16, p. 121])**.** *On peut obtenir un sterne* **S** *à partir d'un oiseau bleu* **B***, d'un cardinal* **C** *et d'un weka* **W**.

Démonstration. À partir de xyz (3 lettres), on cherche à obtenir $xz(yz)$ (4 lettres). Il nous faut donc un oiseau répétiteur (typiquement **W**) pour passer de xyz à, par exemple, $xyzz$ et un autre qui soit compositeur (typiquement **B**). Si **W** s'applique à deux lettres, il suffit de le combiner deux fois avec **B** pour qu'il s'applique à quatre lettres : $\mathbf{B}(\mathbf{BW})txyz = (\mathbf{BW})(tx)yz = \mathbf{BW}(tx)yz = \mathbf{W}((tx)y)z = ((tx)y)z = txyzz$. On cherche un sterne, donc il faudrait que t appliqué à $xyzz$ donne $xz(yz)$ pour tous x, y, z. Il suffit d'utiliser **BBC** à la place de t : $\mathbf{BBC}xyzz = \mathbf{B}(\mathbf{C}x)yzz = (\mathbf{C}x)(yz)z = \mathbf{C}x(yz)z = xz(yz)$. Donc $\mathbf{B}(\mathbf{BW})(\mathbf{BBC})xyz = xz(yz)$ pour tous x, y, z, ce qui est le comportement du sterne.

Définition 28. *Un oiseau* A *est* ***combinatoire d'ordre*** n *(pour* $n \geqslant 1$*) si* n *est le plus petit nombre tel que* $AX_1 \cdots X_n$ *puisse être écrit en tant que terme contenant uniquement des éléments de* $\{X_1, \ldots, X_n\}$ *et des parenthèses.*

Ainsi, **M** et **I** sont combinatoires d'ordre 1 ($\mathbf{M}x = xx, \mathbf{I}x = x$), **K**, **T** ou **WT** sont combinatoires d'ordre 2 ($\mathbf{K}xy = x, \mathbf{T}xy = yx$, etc.), **S**, **B** ou **RRR** d'ordre 3 ($\mathbf{S}xyz = xz(yz), \mathbf{B}xyz = x(yz)$, etc.), et ainsi de suite. La condition de minimalité est nécessaire : par exemple, pour se « débarrasser » de **K** il faut au moins *deux* applications, ainsi $\mathbf{K}x$ ne peut être écrit en tant que terme contenant uniquement x.

Définition 29. *Un oiseau* A *est* ***dérivable*** *d'un ensemble d'oiseaux* $\{O_1, \ldots, O_n\}$*, s'il existe un* $m \geqslant 0$ *et un terme* O *construit en utilisant des éléments de* $\{O_1, \ldots, O_n\}$ *et des parenthèses, tels que* $Ox_1 \cdots x_m = Ax_1 \cdots x_m$ *pour tous* $x_1, \ldots, x_m$.

Ainsi on a vu dans la prop. 27 qu'un sterne **S** est dérivable de $\{\mathbf{B}, \mathbf{C}, \mathbf{W}\}$ puisque $\mathbf{B}(\mathbf{BW})(\mathbf{BBC})$ a le même comportement que **S**, avec $m = 3$.

TABLEAU RÉCAPITULATIF DES COMBINATEURS				
$\mathbf{I}x \triangleright x$	$\mathbf{K}xy \triangleright x$	$\mathbf{S}xyz \triangleright xz(yz)$	$\mathbf{M}x \triangleright xx$	$\mathbf{L}xy \triangleright x(yy)$
$\mathbf{B}xyz \triangleright x(yz)$	$\mathbf{W}xy \triangleright xyy$	$\mathbf{C}xyz \triangleright xzy$	$\mathbf{T}xy \triangleright yx$	$\mathbf{R}xyz \triangleright yzx$

Théorème 30 ([16, p. 170])**.** *Tout oiseau combinatoire est dérivable de* $\{\mathbf{S},\mathbf{K}\}$ *ainsi que de* $\{\mathbf{I},\mathbf{B},\mathbf{C},\mathbf{W},\mathbf{K}\}$.

Il s'agit d'un résultat fondamental de la logique combinatoire que nous allons démontrer dans la section suivante. Il est extrêmement puissant : imaginons que $Xx_1\cdots x_n$ est égal à un terme utilisant uniquement $x_1,\ldots,x_n$; alors quelle que soit la complexité de ce terme, X peut se réécrire avec uniquement des **S**, des **K** et des parenthèses.

V Aspects mathématiques

Nous allons maintenant quitter la forêt enchantée et considérer les aspects mathématiques des notions rencontrées. Partons d'un ensemble dénombrable E et $V(E) = \{x,y,z,u,v,w,\ldots\}$ un ensemble de variables sur E, et formons, d'après la déf. 4, l'ensemble $T(E,V(E))$ des termes sur E et $V(E)$ muni de la règle de simplification de notation des termes.

Définition 31 ([2, p. 6])**.** *Un élément* $\mathbf{Z}$ *de* $T(E,V(E))$ *est appelé un* ***combinateur*** *s'il possède : une* ***arité*** $n_{\mathbf{Z}} \in \mathbb{N}$ *et un* ***axiome***

$$\mathbf{Z}x_1\cdots x_{n_{\mathbf{Z}}} \triangleright M(x_1,\ldots,x_{n_{\mathbf{Z}}})$$

où $x_1,\ldots,x_{n_{\mathbf{Z}}}$ *sont des variables et* $M(x_1,\ldots,x_{n_{\mathbf{Z}}})$ *est un terme sur* $V(E)$ *contenant uniquement des variables parmi* $x_1,\ldots,x_{n_{\mathbf{Z}}}$ *et des parenthèses. L'arité est* le plus petit *entier* $n_{\mathbf{Z}}$ *pour lequel le combinateur* $\mathbf{Z}$ *possède un axiome* $\mathbf{Z}x_1\cdots x_{n_{\mathbf{Z}}} \triangleright M(x_1,\ldots,x_{n_{\mathbf{Z}}})$.

Le lecteur doit considérer les définitions d'oiseaux combinatoires $\mathbf{M},\mathbf{K},\mathbf{I},\mathbf{L},\mathbf{B},\mathbf{W},\mathbf{C},\mathbf{S},\mathbf{T},\mathbf{R}$ que nous avons donné dans la section précédente comme des axiomes de combinateurs représentés par les mêmes symboles. Le nombre de lettres utilisé dans chaque définition sera l'arité du combinateur correspondant. Ainsi, par exemple, le combinateur **S** (qui correspond au sterne dans le monde des oiseaux) est d'arité 3 et son axiome est $\mathbf{S}xyz \triangleright xz(yz)$. Le fait qu'on utilise dans ce texte un petit nombre de lettres majuscules pour noter certains combinateurs ne doit pas occulter le fait qu'on peut en définir une infinité, tous plus complexes les uns que les autres. D'autre part, comme le souligne [8, p. 22], il ne faut pas confondre la notion d'arité de combinateur avec celle d'arité d'une fonction à plusieurs arguments : dire qu'une fonction f est d'arité 3 signifie qu'elle nécessite impérativement trois arguments $f(x,y,z)$ et que l'écriture $f(x,,)$ est syntaxiquement incorrecte — dans le cas des combinateurs, les expressions $\mathbf{S}x$ et $\mathbf{S}xy$ représentent des termes parfaitement valables, et dire que **S** est d'arité 3 signifie qu'il faut *au moins* trois applications (notées xyz) pour que $\mathbf{S}xyz$ puisse être identifié à un terme contenant uniquement les variables x,y,z (en l'occurrence $xz(yz)$).

Les combinateurs ainsi définis sont appelés ***primitifs***. En formant des termes avec *uniquement* des combinateurs primitifs (et pas de variable !) on obtient des ***combinateurs complexes***, pour lesquels on dispose parfois (mais pas toujours [5]) également d'arités et d'axiomes. Exemple : **KI** est un combinateur complexe d'arité 2 et d'axiome $\mathbf{KI}xy \triangleright y$.

Comme déjà mentionné dans le texte, on identifie cinq types basiques de combinateurs :

1. l'*identité* **I**, qui reproduit son argument ;
2. les *répétiteurs*, comme **M** ou **W**, qui redoublent certains arguments (sans changer leur ordre, sans en omettre et sans ajouter des parenthèses) ;
3. les *éliminateurs*, comme **K**, qui suppriment un ou plusieurs arguments, sans rien ajouter et sans changer l'ordre de ceux qui restent ;
4. les *permutateurs*, comme **C**, qui permutent certains arguments, sans rien ajouter ou supprimer ;
5. les *compositeurs*, comme **B**, qui insèrent des parenthèses.

Il nous faut maintenant définir avec plus de rigueur le mécanisme de *réduction* de terme, c'est-à-dire de passage progressif de termes avec combinateurs et variables à des termes avec variables uniquement. Notre stratégie est de commencer par la gauche et selon l'arité de chaque combinateur primitif de chercher à lui fournir le bon nombre d'arguments.

Définition 32 ([2, def. 1.3.1])**.** *Soit* $\mathbf{Z}$ *un combinateur primitif d'arité* n *et* $M_1,\ldots,M_n$ *des termes, alors* $\mathbf{Z}M_1\cdots M_n$ *est appelé un* ***rédex****, de* ***tête*** $\mathbf{Z}$ *et d'****arguments*** $M_1,\ldots,M_n$.

Le mécanisme consistera à chercher à chaque étape le rédex le plus à gauche et à se servir de l'axiome de la tête du rédex pour réarranger les arguments de celui-ci, en omettant celle-là :

5. L'exemple typique est **WWW**, qui ne peut être réduit puisqu'on retrouve **WWW** dès qu'on applique la définition du premier **W**.

Définition 33 ([2, def. 1.3.2]). *Soit* $\mathbf{Z}M_1 \cdots M_n$ *le rédex le plus à gauche de notre terme, et soit* $\mathbf{Z}x_1 \cdots x_n \rhd P(x_1,\ldots,x_n)$ *l'axiome de* $\mathbf{Z}$. *Alors la* ***réduction à une étape*** *consiste à remplacer le rédex par* $P(M_1,\ldots,M_n)$.

Exemple 34. Prenons le terme $\mathbf{BBC}xyzu$. Comme $\mathbf{B}$ est d'arité 3, le premier rédex à gauche est $\mathbf{BBC}x$ (M_1 est $\mathbf{B}$, M_2 est $\mathbf{C}$ et M_3 est x). Rappelons que l'axiome de $\mathbf{B}$ est $\mathbf{B}x_1x_2x_3 \rhd x_1(x_2x_3)$, donc $P(x_1,x_2,x_3) = x_1(x_2x_3)$. On remplace x_1 par $\mathbf{B}$, x_2 par $\mathbf{C}$ et x_3 par x, et donc la première étape donne $\mathbf{BBC}xyzu \rhd \mathbf{B}(\mathbf{C}x)yzu$. Les étapes suivantes donnent

$$\mathbf{BBC}xyzu \rhd \mathbf{B}(\mathbf{C}x)yzu \rhd \mathbf{C}x(yz)u \rhd xu(yz). \qquad (1)$$

Comment formaliser le fait d'aller « aussi loin que possible » en effectuant des réductions ? La réduction à une étape est une relation binaire (entre sa partie gauche et sa partie droite), nous allons prendre sa *clôture transitive* [6] :

Définition 35 ([2, def. 1.3.4, 1.3.5]). *La clôture transitive de la réduction à une étape est appelée* ***réduction faible*** *et est notée par* $\rhd_w$. *Quand un terme n'a pas de rédex, on dit qu'il est sous* ***forme normale***.

Exemple 36. Par (1) on a $\mathbf{BBC}xyzu \rhd_w xu(yz)$.

Cette réduction est appelée *faible* parce qu'on n'a pas de garantie qu'elle aboutisse toujours à une forme normale : si dans l'exemple ci-dessus on n'avait que trois variables (donc sans le u final), la dernière étape serait impossible, par faute d'arguments pour constituer un rédex de tête $\mathbf{C}$.

Il faut maintenant redéfinir la notion de dérivabilité (déf. 29) de Smullyan. Par exemple, nous avons dit que « $\mathbf{R}$ peut être obtenu à partir de $\mathbf{C}$ puisque $\mathbf{R}xyz = \mathbf{CC}xyz$ pour tous x,y,z ».

Définition 37 ([2, def. 1.3.6]). *Soit* $\mathbf{Z}$ *un combinateur et* $\mathbf{Z}x_1 \cdots x_n \rhd M$ *son axiome. On dira que le combinateur complexe* $\mathbf{Z}'$ ***définit*** $\mathbf{Z}$ *si* $\mathbf{Z}'x_1 \cdots x_n \rhd_w M$.

Exemple 38. Soit $\mathbf{Z} = \mathbf{R}$ et $\mathbf{Z}' = \mathbf{CC}$. L'axiome de $\mathbf{R}$ est $\mathbf{R}xyz \rhd yzx$. Mettons $\mathbf{CC}$ à la place de $\mathbf{R}$, d'après la prop. 24 on a $\mathbf{CC}xyz \rhd \mathbf{C}yxz \rhd yzx$, qui est bien la partie droite de l'axiome de $\mathbf{R}$.

Définition 39 ([2, def. 1.3.7, 1.3.8]). *On appelle* ***base*** *un ensemble fini de combinateurs primitifs. On dira qu'une base est* ***combinatoirement complète*** *si tout terme peut être réécrit en utilisant uniquement les éléments de celle-ci.*

Nous sommes prêts maintenant à énoncer le théorème fondamental, dont la démonstration est trop longue et technique pour être donnée ici :

Théorème 40 ([2, lem. 1.3.9]). *Les bases* $\{\mathbf{S},\mathbf{K}\}$ *et* $\{\mathbf{I},\mathbf{B},\mathbf{C},\mathbf{W},\mathbf{K}\}$ *sont combinatoirement complètes.*

VI Représentation des fonctions primitives récursives par des combinateurs

Nous avons vu à la section 3 que les fonctions primitives récursives sont définies à partir de leurs ingrédients de base (la fonction zéro, la fonction successeur, les projections) et les deux procédés qui permettent de les engendrer : la composition et la récursion. Pour montrer que toute fonction (primitive récursive) peut être représentée par un combinateur complexe, il suffit de montrer que c'est le cas pour les ingrédients de base, et ensuite de montrer que la composition et la récursion préservent cette propriété.

Notons $\bar{0}$, $\bar{1}$, $\bar{2}$, etc., les représentations des fonctions constantes 0, 1, 2, etc. en tant que combinateurs. Une approche possible est celle de définir $\bar{0}$ comme le combinateur qui envoie xy à y (x disparaît), $\bar{1}$ comme l'identité, $\bar{2}xy$ comme $x(xy)$, $\bar{3}xy$ comme $x(x(xy))$, et ainsi de suite. On peut facilement vérifier que $\bar{0} = \mathbf{KI}$, $\bar{1} = \mathbf{SB}(\mathbf{KI})$, $\bar{2} = \mathbf{SB}(\mathbf{SB}(\mathbf{KI}))$, etc.

Le lecteur l'aura compris : la représentation de la fonction successeur σ est tout simplement $\mathbf{SB}$.

Qu'en est-il des projections ? Il n'y a pas de formule magique qui fournisse tous les π_k^i mais on peut les trouver par des calculs [7] que :

$$\begin{aligned}
\bar{\pi}_2^1 &: \quad \mathbf{K}xy = x \\
\bar{\pi}_2^2 &: \quad \mathbf{KI}xy = y \\
\bar{\pi}_3^1 &: \quad \mathbf{S}(\mathbf{KK})\mathbf{K}xyz = x \\
\bar{\pi}_3^2 &: \quad \mathbf{KK}xyz = y \\
\bar{\pi}_3^3 &: \quad \mathbf{K}(\mathbf{SK})xyz = z \\
\bar{\pi}_4^1 &: \quad \mathbf{S}(\mathbf{K}(\mathbf{S}(\mathbf{KK})\mathbf{K})\mathbf{K}xyzt = x \\
\bar{\pi}_4^2 &: \quad \mathbf{K}(\mathbf{S}(\mathbf{KK})\mathbf{K})xyzt = y \\
\bar{\pi}_4^3 &: \quad \mathbf{K}(\mathbf{KK})xyzt = z \\
\bar{\pi}_4^4 &: \quad \mathbf{K}(\mathbf{K}(\mathbf{SK}))xyzt = t, \text{ etc.}
\end{aligned}$$

Considérons maintenant la composition. Le cas le simple est celui des fonctions d'une variable : $f(x) = g(h(x))$. Si les termes correspondant à f,g,h sont

6. La ***clôture transitive*** d'une relation binaire est basée sur le principe « l'ami de mon ami est mon ami » : si R dénote la relation binaire et R^* sa clôture transitive, alors on a $(a,b) \in R^*$ s'il existe $a = a_1,\ldots,a_n = b$ tels que $(a_i,a_{i+1}) \in R$ pour tout $1 \leqslant i < n$.

7. Le lecteur intéressé trouvera sur http://perso.telecom-bretagne.eu/yannisharalambous/ressources/ un interprète de logique combinatoire programmé en Java par John Tromp, qui sert à ce type de calculs.

$\bar{f}, \bar{g}, \bar{h}$, alors on sait déjà que $\bar{f} = \mathbf{B}\bar{g}\bar{h}$. Prenons le cas où g est une fonction de deux variables et donc $f(x) = g(h_1(x), h_2(x))$. Alors on peut la curryfier pour obtenir $f(x) = g^*(h_1(x))(h_2(x))$ et le combinateur que l'on cherche sera de la forme $\mathbf{Z}xyzt = x(y(t)(z(t)))$, les calculs montrent qu'il s'agit de $\mathbf{B}(\mathbf{BS})\mathbf{B}$. On peut faire de même quelque soit le nombre de variables de g et des h_*.

Dernière opération à traiter : la récursion. Le traitement de celle-ci serait trop long pour cet article (nous renvoyons le lecteur à [2, p. 64–81]), nous allons juste donner un élément de réponse qui joue un rôle central. En effet, dans la procédure de récursion on trouve un test « si telle variable numérique est égale à zéro, alors faire l'étape 1, sinon faire l'étape 2 ». Nous savons déjà que les les valeurs numériques sont modélisées par $\mathbf{KI}$ pour $\bar{0}$ et par des $\mathbf{SB}(\cdots(\mathbf{SB}(\mathbf{KI}))\cdots)$ pour les autres valeurs. Il nous faut donc un $\mathbf{D}xyz$ tel que $\mathbf{D}xy\bar{0}$ donne x et $\mathbf{D}xy\bar{\imath}$ donne y pour tout $i > 0$. Ce $\mathbf{D}$ existe, il est $\mathbf{C}(\mathbf{BC}(\mathbf{B}(\mathbf{CI})\mathbf{K}))$, comme on peut facilement vérifier.

Une fonction primitive récursive est obtenue à partir des fonctions $\bar{0}$, de $\bar{\sigma}$ et des projections, en effectuant des compositions et des récursions. En remplaçant les ingrédients de base et ses deux opérations par les combinateurs correspondant, on obtient une représentation de toute fonction en logique combinatoire, comme annoncé. Celle-ci fonctionne donc comme un langage de programmation qui peut modéliser et calculer les valeurs de toute fonction primitive récursive.

VII L'algèbre de Boole

Mis à part les fonctions primitives récursives, la logique combinatoire peut aussi nous permettre de faire des calculs d'*algèbre de Boole*, c'est-à-dire des calculs avec des variables qui peuvent prendre les valeurs V (vrai) ou F (faux) et avec des opérations telles que la négation NEG, la conjonction AND et la disjonction OR. Ce genre de calcul est très utile dans la conception de circuits électroniques. Notons que cette approche ne prétend en aucun cas atteindre des bonnes performances de calcul — ce qui nous a intéressé ici est la manière de transformer une formule logique en terme de logique combinatoire.

Soit $\mathscr{F}$ une formule de l'algèbre de Boole, c'est-à-dire une formule utilisant :

— des variables $a, b, c, \ldots$ prenant des valeurs dans l'ensemble $\{V, F\}$ (vrai, faux),
— une fonction unaire NEG (définie par $\mathsf{NEG}(V) = F$, etc.), et
— deux fonctions binaires AND et OR (définies par $\mathsf{AND}(V,V) = V$, $\mathsf{OR}(F,V) = V$, etc.).

Si $\mathscr{F}$ possède n variables $a_1, \ldots, a_n$, alors soit $(\hat{a}_1, \ldots, \hat{a}_n)$ une distribution de valeurs V ou F pour les a_i. Alors en remplaçant chaque a_i par sa valeur $\hat{a}_i$ dans $\mathscr{F}$ et en faisant les calculs, on obtient une valeur unique V ou F pour $\mathscr{F}(\hat{a}_1, \ldots, \hat{a}_n)$.

On appelle *table de vérité* de $\mathscr{F}$ l'application qui à chaque $(\hat{a}_1, \ldots, \hat{a}_n)$ associe $\mathscr{F}(\hat{a}_1, \ldots, \hat{a}_n)$. Cette application caractérise $\mathscr{F}$ entièrement.

La question que l'on se pose est : peut-on faire en sorte que la table de vérité de $\mathscr{F}$ puisse être obtenue par un calcul de termes de logique combinatoire ? Formulé autrement : en considérant la logique combinatoire en tant que langage de programmation, peut-on écrire un programme dans ce langage pour chaque formule d'algèbre de Boole ? Ce programme devrait nous fournir les mêmes valeurs que la formule pour toutes les combinaisons de valeurs V et F des variables. La réponse est affirmative :

Théorème 41. *Soit $\mathscr{F}$ une formule de l'algèbre de Boole utilisant les variables $a_1, \ldots, a_n$. Transformons $\mathscr{F}$ en terme M de logique combinatoire, en suivant les règles inductives suivantes :*

1. *les variables de $\mathscr{F}$ deviennent variables de M ;*
2. *à la place de $\mathsf{NEG}(\mathscr{F}')$ (où $\mathscr{F}'$ est une sous-formule de $\mathscr{F}$), écrivons $\mathscr{F}'(\mathbf{SK})\mathbf{K}$;*
3. *à la place de $\mathsf{AND}(\mathscr{F}', \mathscr{F}'')$ (où $\mathscr{F}'$ et $\mathscr{F}''$ sont des sous-formules de $\mathscr{F}$), écrivons $\mathscr{F}'\mathscr{F}''(\mathbf{SK})$;*
4. *à la place de $\mathsf{OR}(\mathscr{F}', \mathscr{F}'')$ (où $\mathscr{F}'$ et $\mathscr{F}''$ sont des sous-formules de $\mathscr{F}$), écrivons $\mathscr{F}'\mathbf{K}\mathscr{F}''$.*

Une fois le terme M obtenu, pour chaque distribution de valeurs $(\hat{a}_1, \ldots, \hat{a}_n)$, remplaçons la variable a_i de M par $\mathbf{K}$ si $\hat{a}_i = V$, et par $(\mathbf{SK})$ si $\hat{a}_i = F$.

Alors M a toujours une forme normale, qui ne peut être que $\mathbf{K}$ ou $\mathbf{SK}$. Dans le premier cas, cela veut dire que $\mathscr{F}(\hat{a}_1, \ldots, \hat{a}_n) = V$, et dans le deuxième que $\mathscr{F}(\hat{a}_1, \ldots, \hat{a}_n) = F$.

Il s'ensuit que les réductions du terme M sont un moyen de calcul de la table de vérité de $\mathscr{F}$.

Notons aussi que, puisque $\mathbf{SK} = \mathbf{KI}$, on retrouve en tant que représentations de « vrai » et de « faux » les combinateurs $\mathbf{KI}$ et $\mathbf{K}$ utilisés dans la section précédente pour les nombres $\bar{0}$ et $\bar{1}$.

Exemple 42. Prenons un exemple : montrons la distributivité à gauche de la conjonction sur la disjonction : $a \wedge (b \vee c) = (a \wedge b) \vee (a \wedge c)$ pour tous $a, b, c \in \{V, F\}$.

Réécrivons la formule en utilisant AND et OR : $\mathsf{AND}(a, \mathsf{OR}(b,c)) = \mathsf{OR}(\mathsf{AND}(a,b), \mathsf{AND}(a,c))$. Prenons la partie gauche : nous avons $\mathsf{AND}(a, \mathsf{OR}(b,c)) = \mathsf{AND}(a, b\mathbf{K}c) = a(b\mathbf{K}c)(\mathbf{SK})$. La partie droite donne $\mathsf{OR}(\mathsf{AND}(a,b), \mathsf{AND}(a,c)) = \mathsf{OR}(ab(\mathbf{SK}), ac(\mathbf{SK})) = ab(\mathbf{SK})\mathbf{K}ac(\mathbf{SK})$. Les termes $a(b\mathbf{K}c)(\mathbf{SK})$ et $ab(\mathbf{SK})\mathbf{K}ac(\mathbf{SK})$ sont différents, mais

qu'à cela ne tienne : ce qui importe ce sont leurs valeurs quand a, b et c prennent les valeurs **K** (pour « vrai ») et (**SK**) (pour « faux »). On a donc 2^3 calculs à faire pour $(a,b,c) = (\mathbf{K},\mathbf{K},\mathbf{K}), (a,b,c) = (\mathbf{K},\mathbf{K},(\mathbf{SK}))$, etc. Faisons le premier et laissons le reste en guise d'exercice : la partie gauche devient $\mathbf{K}(\mathbf{KKK})(\mathbf{SK}) = \mathbf{KKK} = \mathbf{K}$. La partie droite devient $\mathbf{KK}(\mathbf{SK})\mathbf{K}(\mathbf{KK}(\mathbf{SK})) = \mathbf{KK}(\mathbf{KK}(\mathbf{SK})) = \mathbf{K}$, ce qu'il fallait démontrer.

VIII Applications à la linguistique mathématique

Prenons la phrase *Gérard dort*. Comme nous l'avons vu dans notre précédent article [11], on peut traduire cette phrase dans le cadre de la logique du premier ordre par la formule « dort(Gérard) », où « dort » est un prédicat unaire (c'est-à-dire une fonction à un argument qui prend des valeurs dans l'ensemble {vrai, faux}) et « Gérard » est une constante.

Nous venons juste de dire que « dort » est une fonction, donc « dort(Gérard) » est l'*application* de la fonction « dort » à la constante « Gérard ». Le lecteur perspicace se doute déjà du fait que s'il est question d'« application de fonction », c'est bien parce qu'on a l'intention de se servir de logique combinatoire. Effectivement, on peut considérer qu'on applique le verbe « dort » à son sujet « Gérard », pour obtenir un terme « dort(Gérard) » qui est considéré comme une phrase complète.

De même, si on prend la phrase *Gérard aime Alice*, on peut d'abord appliquer le verbe à son COD pour obtenir un groupe verbal « aime(Alice) », et ensuite appliquer ce groupe verbal au sujet pour obtenir le terme « (aime(Alice))(Gérard) » qui représente la phrase dans sa totalité.

Ceux qui ont lu [11] observeront que ce n'est pas de cette manière que l'on y a traité la même phrase : on a plutôt représenté « aime » par un prédicat binaire « aime(Gérard,Alice) » dont les constantes « Gérard » et « Alice » étaient les arguments. En réalité les deux approches sont équivalentes : au lieu de traiter les deux arguments « simultanément » on prend d'abord le premier, et cela nous donne une fonction qu'on applique au deuxième, ce procédé est appelé *curryfication* en référence au fondateur de la logique combinatoire Haskell Curry.

Il ne reste plus qu'à faire le lien entre la phrase écrite en langage naturel *Gérard aime Alice* et le terme qui va la représenter. On constate que « dans la vraie vie », le COD se trouve à droite du verbe et le sujet à gauche [8], il faudrait donc que l'application puisse se faire dans un sens ou dans l'autre.

Cela est proposé dans une technique de représentation de la syntaxe introduite par M. Steedman [21], cette approche est appelée CCG (*Grammaires catégorielles combinatoires*) et se base, comme son nom l'indique, sur la logique combinatoire.

Il s'agit de logique combinatoire typée, dans le sens où on attache à chaque variable et à chaque combinateur un type (qui correspond à une fonction grammaticale), et que seules les applications entre types compatibles sont autorisées.

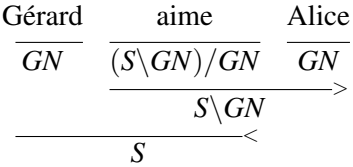
L'idée de Steedman est la suivante : il attache à chaque mot une catégorie syntaxique représentée par une variable typée, les types étant : groupe nominal *GN*, groupe verbal *GV*, phrase *S*, etc. Ensuite il utilise une notation spécifique pour écrire les termes : cette notation dévoile à quel type de terme on peut l'appliquer et de quel type est le terme qui en résulte.

Prenons un exemple : *Gérard dort*. Le type de *Gérard* va tout simplement être *GN* (groupe nominal). Quel va être le type du verbe *dort* ? Il faut que ce type exprime (a) le fait qu'il va s'appliquer à un groupe nominal *GN* et (b) que ce faisant il va fournir une phrase *S*. Mais, attention, on voit bien que *Gérard* se trouve à gauche du verbe : Steedman introduit cette information dans la notation du type du verbe en utilisant la notation $S\backslash GN$ (la barre oblique $\backslash$ indique que, dans la phrase, le terme à qui $S\backslash GN$ va s'appliquer sera à sa gauche). En appliquant $(S\backslash GN)(GN)$ on obtient un terme de type *S*, le type de la phrase. Si le sujet du verbe était à droite, il aurait suffi d'écrire S/GN.

Le cas d'un verbe transitif illustre parfaitement le génie de l'approche : prenons *Gérard aime Alice*, dont le groupe verbal est *aime Alice*. Il suffit de modéliser *aime* par $(S\backslash GN)/GN$ ce qui signifie qu'on va d'abord l'appliquer à un groupe nominal qui se trouve immédiatement à droite (le COD), et que cela nous fournira un $S\backslash GN$ (l'équivalent d'un verbe intransitif comme « dort » dans l'exemple précédent), que l'on appliquera à un *GN* se trouvant à gauche (le sujet) pour obtenir une phrase *S*. Ainsi, la modélisation des groupes nominaux est indépendante de leur position dans la phrase : ils sont tous de type *GN*. C'est le verbe qui contient l'information sur son comportement, sur le type de compléments dont il a besoin et sur leurs positions relatives à lui.

On a donc une *application à gauche* (« application » au sens des termes de la logique combinatoire) ainsi qu'une *application à droite*, et on représente la structure de la phrase comme ceci :

8. Dans des langues comme l'arabe, le russe, le latin ou le grec, l'ordre des constituants de la phrase est plus libre. Ces langues lèvent l'ambiguïté sur le rôle syntaxique de chaque groupe nominal par l'utilisation de cas (nominatif, accusatif, etc.), ainsi, sans changer l'ordre des mots on peut écrire *amat virgo rosam* (la fille aime la rose) ainsi que *amat virginem rosa* (la rose aime la fille).



Le symbole qui se trouve en fin de filet horizontal indique le type d'opération qui permet de passer de la partie au-dessus du filet à celle au-dessous. Le $>$ indique une application à droite, et le $<$ une application à gauche. Les règles de production de la grammaire ainsi définie sont :

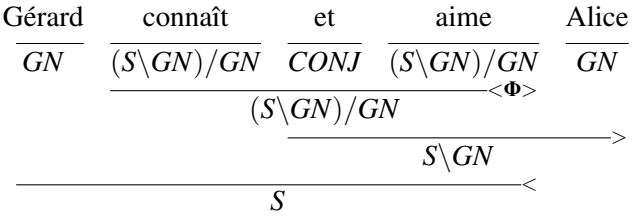
$$X/Y \quad Y \rightarrow X \qquad (>)$$
$$Y \quad X\backslash Y \rightarrow X, \qquad (<)$$

c'est-à-dire que quelques soient les catégories des termes, du moment qu'elles sont compatibles on peut appliquer X/Y à Y pour obtenir un terme de type X, ou $X\backslash Y$ à Y pour obtenir de nouveau X.

Un phénomène syntaxique très courant est la *coordination*, comme dans *Gérard connaît et aime Alice*. Selon le Grevisse [9, § 269], on a affaire à un impératif d'économie : « quand, dans les termes coordonnés, il y a des éléments identiques, la tendance naturelle est de ne pas répéter ces éléments communs ». L'approche de Steedman pour garantir la symétrie des deux verbes est d'introduire une règle

$$X \quad CONJ \quad X' \rightarrow X'', \qquad (<\mathbf{\Phi}>)$$

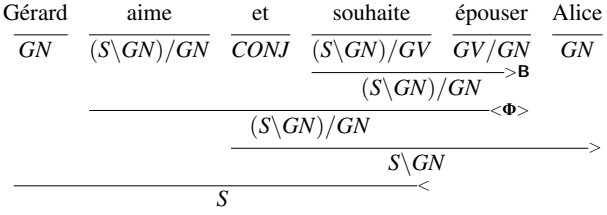
où $CONJ$ est la catégorie de la particule de conjonction *et*, et le fait d'utiliser la même lettre X signifie que X, X' et X'' doivent être du même type (c'est une règle de base de la coordination : on ne peut coordonner que des éléments de même statut [9, § 260]). Cela donne :



Prenons maintenant la phrase *Gérard aime et souhaite épouser Alice*. Ici, *souhaite épouser* agit comme un seul verbe. Steedman considère que les deux verbes « souhaiter » et « épouser » se composent en tant que combinateurs. Il donne à *souhaite* la catégorie $(S\backslash GN)/GV$ (autrement dit : on prend un verbe et on retourne un $S\backslash GN$) et à *épouser* (qui, rappelons-le, est à l'infinitif) la catégorie GV/GN (donc : il prend un GN et retourne un GV). En composant les deux, on obtient un verbe de la même catégorie que *aime* : $(S\backslash GN)/GN$ et à partir de ce moment l'analyse est la même qu'avant. Puisqu'il s'agit de composition de combinateurs, la règle de grammaire correspondante est celle de l'oiseau bleu **B** :

$$X/Y \quad Y/Z \rightarrow_{\mathbf{B}} X/Z, \qquad (>\mathbf{B})$$

et voici comment analyser la phrase :



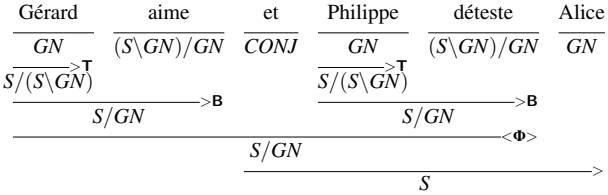
Pour finir, un autre cas de coordination avec impératif d'économie : *Gérard aime et Philippe déteste Alice*. La première partie *Gérard aime* n'est pas directement adjacente à *Alice*, qu'il doit partager en tant que COD avec *Philippe déteste*. On va faire la chose suivante : au lieu de traiter d'abord le COD pour passer de $(S\backslash GN)/GN$ (verbe) appliqué à GN (COD), pour obtenir $S\backslash GN$, on va d'abord traiter le sujet (qui est différent pour chacune des propositions).

Mais comment accéder au sujet alors que la catégorie du verbe est $(S\backslash GN)/GN$ et le sujet y est bien profondément caché ? Il est évident que l'on ne peut pas appliquer cette catégorie à quelque chose se trouvant à gauche (puisqu'elle s'attend à recevoir un GN venant de la droite). On va donc de nouveau se servir de composition. Pour trouver avec quoi composer $(S\backslash GN)/GN$, posons-nous la question de ce que nous voulons obtenir : on cherche quelque chose qui, appliqué au COD (qui est GN) nous donne S. On cherche donc un S/GN. Avec quoi composer $(S\backslash GN)/GN$ pour obtenir S/GN ? Il n'y a qu'une réponse possible : $S/(S\backslash GN)$. Il faut donc remplacer le GN du sujet par $S/(S\backslash GN)$.

Ce que nous venons de décrire est une opération standard, appelée ***montée de type*** (*type raising* en anglais) et c'est d'ailleurs elle qui a inspiré la lettre du combinateur **T** (le tarin).

Une fois la montée de type effectuée, la situation se renverse : c'est le sujet ($S/(S\backslash GN)$) qui se compose avec le verbe ($(S\backslash GN)/GN$) pour produire un groupe verbal spécial (S/GN), qui va chercher à s'appliquer à droite, c'est-à-dire au COD (GN), pour produire enfin la phrase S.

Voici comment analyser la phrase :



IX La plus petite base possible

Le lecteur pense sans doute que $\{\mathbf{S},\mathbf{K}\}$ est la plus petite base possible pour générer l'ensemble des combinateurs. Eh bien, on peut faire encore mieux ! Chris Barker a introduit en 2001 [1] la base $\{\iota\}$ (le iota étant la plus petite lettre grecque, d'où l'expression « je ne bouge pas d'un iota »), définie comme suit :

$$\iota x := x\mathbf{SK} \text{ pour tout } x.$$

Pour retrouver $\mathbf{I},\mathbf{K}$ et $\mathbf{S}$, il suffit d'écrire :

$$\mathbf{I} = \iota\iota, \quad \mathbf{K} = \iota(\iota(\iota\iota)), \quad \mathbf{S} = \iota(\iota(\iota(\iota\iota))).$$

Vérifions la première : $\iota\iota x = \iota\mathbf{SK}x = \mathbf{SSKK}x = \mathbf{SK}(\mathbf{KK})x = \mathbf{K}x(\mathbf{SK}x) = x = \mathbf{I}x$, pour tout x (les deux autres égalités sont laissées en guise d'exercice). Comme on peut obtenir $\mathbf{S}$ et $\mathbf{K}$ à partir de ι, cela veut bien dire que $\{\iota\}$ est une base, et on ne peut faire plus petit puisqu'elle ne comporte qu'un seul élément.

X Unlambda : un langage de programmation ésotérique

Croyez-le ou pas, il existe une communauté de personnes qui s'amuse à inventer des langages de programmation rocambolesques, pour tester les limites de l'ordinateur et... des cerveaux de leurs utilisateurs. Ces langages sont appelés *ésotériques* et le langage *Unlambda* de David Madore (1999) en fait partie.

L'idée est simple : si on peut tout faire avec la base de combinateurs $\{\mathbf{S},\mathbf{K}\}$ (auxquels on ajoute le $\mathbf{I}$ pour le plaisir), pourquoi ne pas les utiliser directement pour écrire des programmes ? C'est ce que fait Unlambda.

La syntaxe en est on ne peut plus simple : on écrira `s` pour $\mathbf{S}$, `k` pour $\mathbf{K}$, `i` pour $\mathbf{I}$ et `` ` `` (accent grave) pour l'application. D'autre part, comme on veut bien avoir une sortie pour vérifier le bon fonctionnement des programmes, Unlambda possède des combinateurs unaires `.a` (resp. `.b`, `.c`, etc.), qui s'appliquent à ce qui suit, et ce faisant, envoient la lettre `a` (resp. `b`, `c`, etc.) à la sortie. Ainsi que le combinateur unaire `r` pour envoyer un retour-chariot à la sortie.

Notons qu'Unlambda n'utilise pas de parenthèses : `` `ki `` signifie que `k` s'applique à `i` (c'est-à-dire $\mathbf{KI}$) ; `` ``ski `` signifie que `s` s'applique à `k` et le terme qui en résulte s'applique à `i` (il s'agit donc de $\mathbf{SKI}$) ; `` `s`ki `` par contre signifie (en lisant de droite à gauche) qu'il faut appliquer `k` à `i` et ensuite `s` au terme qui en résulte (ce qui correspond $\mathbf{S}(\mathbf{KI})$).

Pour mieux comprendre, transcrivons $\mathbf{S}(\mathbf{KKI})(\mathbf{KI})(\mathbf{KI})$: $\mathbf{KI}$ est tout simplement `` `ki ``, et $\mathbf{KKI}$ s'écrit `` ``kki `` (autrement dit : pour écrire une chaîne de $n$ symboles sans parenthèses, il faut $(n-1)$ accents graves suivis des symboles Unlambda en question). On a donc trois « objets » `` ``kki ``, `` `ki `` et `` `ki ``, et on veut leur appliquer `s`, de combien d'accents graves faut-il le précéder ? $\mathbf{S}$ est ternaire et il est ici suivi de trois éléments, cela fait 4 « objets » en tout, donc $4-1=3$ accents graves : `` ```s``kki`ki`ki `` est le résultat souhaité.

Pour écrire `Quadrature`, il nous faut donc le programme suivant : `` `r``````````.Q.u.a.d.r.a.t.u.r.ei `` (le `i` à la fin ne servant qu'à fournir un argument au terme unaire qui le précède).

Et comme le lecteur peut s'en convaincre, non pas bien sûr en le lisant, mais en l'exécutant, le programme :

```
```s``s``sii`ki`k.*``s``s`ks``s`k`s`ks``s
``s`ks``s`k`s`kr``s`k`sikk`k``s`ksk
```

écrira des lignes avec des astérisques, dont les longueurs sont égales aux termes successifs de la *suite de nombres de Fibonacci* ! Le lecteur courageux trouvera sur [13] des explications, ainsi que des interpréteurs du langage Unlambda.

Exercice 1. Décrire un langage de programmation qui n'utilise qu'un seul combinateur, inspiré du ι de Barker.

XI Historique

Le premier à avoir introduit les notions fondamentales de logique combinatoire, en 1920, a été le mathématicien russe Moiseï Èliévitch Schönfinkel (1889–1942), qui, après avoir écrit deux articles, a été interné en asile psychiatrique en 1927, et est mort à Moscou pendant la guerre (il avait écrit d'autres articles mais ses voisins les ont utilisés pour se chauffer...). Son but était de proposer une logique aussi puissante que la logique du premier ordre, mais *sans variables ni quantificateurs* (!). C'était également la première fois que la *fonction* (plutôt que l'*ensemble*) devenait l'objet primitif d'une théorie logique. Les idées de Schönfinkel ont inspiré Haskell Curry (1900–1982), un doctorant de Hilbert, qui a été parmi les plus importants logiciens du XX$^{\text{e}}$ siècle. Le très populaire langage de programmation *Haskell*, datant de 1998, est nommé en référence au prénom de Curry. À la même époque, un autre Américain, Alonzo Church (1903–1995), inventait le λ-calcul, qui utilise des termes de manière similaire à la logique combinatoire, mais également des variables liées, à travers l'opération d'*abstraction* [11, V.2]. Parmi les doctorants de Church on trouve Raymond Smullyan (1919–2017) à qui nous devons la forêt enchantée : il aurait choisi l'allégorie des oiseaux parce que Curry était grand amateur d'oiseaux, il les observait à partir de l'âge de 15 ans, ce qui lui a valu une collection de pas
```

moins de 150 000 fiches d'observation un demi-siècle plus tard [9].

XII Conclusion

Dans cet article, après avoir défini les fonctions primitives récursives, nous avons donné une introduction à la logique combinatoire à travers l'approche ludique de Smullyan [16] ainsi qu'une autre, plus rigoureuse, et nous avons montré comment les combinateurs de la logique combinatoire peuvent représenter toute fonction primitive récursive. Ensuite nous avons fourni deux applications de la logique combinatoire : la première — classique — à l'algèbre booléenne, et la deuxième — plus inattendue — à l'analyse syntaxique du langage naturel. Nous espérons que cette escapade aux confins des mathématiques, de la logique et de la linguistique a diverti le lecteur et lui a donné envie d'investiguer davantage ces domaines où il y a encore tant à découvrir et à explorer.

Références

[1] Chris BARKER : Iota and jot : the simplest languages ? *The Esoteric Programming Languages Webring*, 2001. https://perma.cc/M6US-33MA.

[2] Katalin BIMBÓ : *Combinatory Logic, pure, applied and typed*. Discrete Mathematics and its Applications. CRC Press, 2012.

[3] Jean-Pierre DESCLÉS, Gaëll GUIBERT et Benoît SAUZAY : *Logique combinatoire et λ-calcul : des logiques d'opérateurs*. Cépaduès, 2016.

[4] Jean-Pierre DESCLÉS : Théorème de Church-Rosser et structuration des langues naturelles. *Mathématiques et sciences humaines*, 103:67–92, 1988.

[5] Jean-Pierre DESCLÉS : Combinatory logic, language, and cognitive representations. *In* P. WEINGARTNER, éditeur : *Alternative Logics. Do Sciences Need Them ?*, pages 115–148. Springer, 2004.

[6] Robert FEYS : La technique de la logique combinatoire. *Revue philosophique de Louvain*, 44: 74–103, 1946. http://www.persee.fr/doc/phlou_0035-3841_1946_num_44_1_4039.

[7] Jean-Pierre GINISTI : Présentation de la logique combinatoire en vue de ses applications. *Mathématiques et sciences humaines*, 103:45–66, 1988. http://www.numdam.org/item?id=MSH_1988__103__45_0.

[8] Jean-Pierre GINISTI : *La logique combinatoire*, volume 3205 de *Que sais-je ?* PUF, 1997.

[9] Maurice GREVISSE et André GOOSSE : *Le bon usage*. De Boeck, 16e édition, 2016.

[10] Chris HANKIN : *Lambda Calculi. A Guide for Computer Scientists*, volume 3 de *Graduate Texts in Computer Science*. Clarendon Press, 1994.

[11] Yannis HARALAMBOUS : Les mathématiques de la langue : l'approche formelle de Montague. *Quadrature*, 98:9–19, 2015.

[12] Roger J. HINDLEY et Jonathan P. SELDIN : *Lambda-Calculus and Combinators. An Introduction*. Cambridge University Press, 2008.

[13] David MADORE : The Unlambda programming language. http://www.madore.org/~david/programs/unlambda/.

[14] Cyril NICAUD : Ordres et structures inductives. http://www-igm.univ-mlv.fr/~nicaud/poly/L2_2014.pdf, 2014.

[15] György E. RÉVÉSZ : *Lambda-Calculus, Combinators, and Functional Programming*. Cambridge University Press, 1988.

[16] Raymond SMULLYAN : *To mock a mockingbird, and other logic puzzles including an amazing adventure in combinatory logic*. Knopf, 1985.

[17] Raymond SMULLYAN : *Juegos para imitar a un pájaro imitador*, volume 15 de *Colleción Libertad y Cambio, Serie Juegos*. Editorial Gedisa, 1989.

[18] Raymond SMULLYAN : *Oponašati oponašalko : logične uganke in presenetljiva pustolovščina v globine kombinatorne logike*. Logika, 1991.

[19] Raymond SMULLYAN : *Spottdrosseln und Metavögel, Computerrätsel, mathematische Abenteuer und ein Ausflug in die vogelfreie Logik*. Wolfgang Krüger Verlag, 1999.

[20] Raymond SMULLYAN : *A csúfolórigó nyomában, egy lebilincselő kaland a kombinatorikus logika világában*. A logika világa. Typotex, 2012.

[21] Mark STEEDMAN : *The syntactic process*. The MIT Press, 2000.

[22] Mark STEEDMAN et Jason BALDRIDGE : Combinatory categorial grammar. *In* Robert D. BORSLEY et Kersti BÖRJARS, éditeurs : *Non-Transformational Syntax*, pages 181–224. Wiley-Blackwell, 2011.

[23] Pierre WOLPER : *Introduction à la calculabilité*. Dunod, 3e édition, 2006.

9. Cf. https://perma.cc/MHR6-Q796.