1

Les Expressions Préfixées

1.1 Les niveaux de langage dans RACKET

Le logiciel RACKET (`www.racket-lang.org`) contient tous les outils pour programmer en SCHEME. L'équipe qui lui a donné naissance et qui le maintient a de multiples intérêts : recherche fondamentale (de nombreuses thèses chaque année sur des sujets en général assez théoriques, de la sémantique des langages aux technologies du Web), produits logiciels, mais aussi l'enseignement car les bons chercheurs se sont toujours fortement impliqués dans l'enseignement de leur discipline. Tous engagés dans la formation des étudiants d'un bout à l'autre du cursus universitaire (voire scolaire pour certains), ils ont été amenés à offrir des *niveaux de langage* à l'intérieur du système de développement, de manière à ne pas plonger trop tôt le débutant dans une complication, une généralité et une puissance inutiles.

Parmi les langages proposés (tous basés *in fine* sur SCHEME), les spécialistes trouveront le langage ALGOL-60 pour les nostalgiques, le langage FRTIME pour la programmation réactive, un LAZY RACKET pour la programmation paresseuse, un TYPED RACKET avec des annotations de type, etc. Mais surtout en ce qui nous concerne des **langages d'enseignement** utilisés avec le livre pédagogique introductif *How to Design Programs* [HtDP]. Le premier niveau est *Beginning Student*, qui limite le langage à un noyau très réduit, purement fonctionnel. D'autres niveaux suivent, et nous opterons quant à nous, dans cette première partie, pour le dernier niveau **Étudiant niveau avancé**, même si nous n'utiliserons pas toutes les fonctionnalités de ce niveau. Pour faire ce choix, il suffit d'aller dans le menu *Langages*, de choisir *Sélectionner le langage*, de dérouler l'onglet *How to Design Programs*, pour choisir *Étudiant niveau avancé*. Demandez aussi à *Montrer les détails*, et cochez en plus les cases *write* et *Fractions mêlées*.

Si vous êtes curieux, sachez que le niveau *Étudiant niveau avancé* contient à la fois une partie purement fonctionnelle et une partie impérative. Nous nous restreindrons à une programmation fonctionnelle dans cette première partie du livre, mais nous au-

rons besoin de quelques petits compléments, comme la forme séquentielle `(begin ...)`, les fonctions prenant un nombre quelconque d'arguments, et la fonction `printf` pour afficher des résultats. Bref quelques petits agréments cosmétiques débordant du cadre strict et puriste de la programmation fonctionnelle, mais qui rendent la vie plus facile. Bien entendu, nous nous interdirons d'utiliser les opérateurs de mutation (affectation, modification des composantes d'une structure ou d'un vecteur). Ceci sera abordé dans la seconde partie dédiée à la programmation impérative.

Puisque nous en sommes aux réglagles, allez dans le menu *DrRacket* jeter un œil aux **Préférences**. Je vous donne mes choix préférés, mais vous ferez comme bon vous semble, l'ergonomie ne se discute pas...

- *Police.* Prenez une police de taille fixe : Monaco-18 sur Mac, Courier-18 sur Windows. Les écrans sont suffisamment larges pour ne pas nous blesser les yeux !
- *Couleurs.*
  - *Couleurs d'arrière-plan.* J'ai opté pour un jaune très pâle pour la couleur d'arrière plan, le blanc s'avérant fatiguant à la longue. Pour le surlignage des parenthèses, j'ai pris du vrai jaune plutôt que du gris. La couleur du texte, essentiellement de la bannière d'accueil au toplevel, sera en noir.
  - *Scheme.* Les *symboles* seront en noir, les *mots réservés* en bleu gras, les *commentaires* en marron sombre, les *chaînes de caractères* en vert sombre, les *constantes* en noir, les *parenthèses* en rouge, les *erreurs* en rouge gras, et les *autres* en noir.

Encore une fois, ces choix sont personnels. Il m'arrive d'en changer et de travailler sur fond noir. Prenez le temps de regarder les menus de DrRacket, ils contiennent beaucoup de choses intéressantes dont nous ne parlerons pas. À propos de DrRacket : ce nom est en réalité celui du logiciel de développement intégré de Racket comprenant un éditeur de textes, un débogueur, un profileur de code pour avoir des statistiques sur les fonctions les plus souvent utilisées, etc. Néanmoins, personne ne vous oblige à utiliser DrRacket. Il est aussi possible de travailler avec Racket de manière beaucoup plus spartiate, au sein d'un éditeur de texte comme Emacs. Ce sont précisément les langages de type Lisp (en particulier InterLisp) qui ont initié les plus grosses avancées dans l'ergonomie des environnements de programmation. L'éditeur Emacs est d'ailleurs entièrement écrit en Lisp, ce qui ne l'empêche pas d'être utilisé pour programmer en C.

Le véritable moteur de Racket se nomme `racket` (anciennement `mzscheme`). Il est invocable directement sur la ligne de commande. Un gros intérêt de cette possibilité (pour le programmeur averti) est de pouvoir rédiger en Racket ses propres **scripts**, pour la ligne de commande Unix par exemple [§ 13.6.3].

```
1 ~/Livre/$ racket                    ; dans le répertoire bin de Racket
2 Welcome to Racket v5.0 [3m]
3 >
```

1.2 La notation préfixée complètement parenthésée

Notre bonne vieille arithmétique utilise principalement la notation dite *infixée*, dans laquelle l'opérateur est placé entre ses deux opérandes, comme dans $2 + 3$. Un long apprentissage nous a habitué à décoder mentalement des expressions ambigües contenant plusieurs opérateurs, par exemple :

```
2+3x/4-5
```

Bien parenthésée, cette expression devrait s'écrire :

```
(2+((3x)/4))-5
```

Si nous ne sommes pas génés par l'absence de parenthèses, c'est que nous possédons de l'information supplémentaire sur les *priorités* des opérateurs. Nous savons par exemple qu'une multiplication a priorité sur une addition, et que l'expression $2 + 3x$ signifie bien 2 additionné à la multiplication de 3 par x.

Le langage mathématique associe donc à chaque opérateur un *poids* plus ou moins élevé représentant sa priorité. Cette association fait partie du non-dit de la pratique mathématique, que l'informatique et la nécessaire rigueur grammaticale de ses langages de programmation a contribué à rendre explicite. D'autant que les mathématiciens se permettent d'utiliser concurremment d'autres notations, par exemple *préfixée* dans $f(x, g(y))$ qui voit l'opérateur précéder ses opérandes, ou *postfixée* dans laquelle l'opérateur suit ses opérandes, comme dans la notation $n!$, sans compter les expressions mixtes utilisant jusqu'aux trois notations en même temps, comme $f(x + 1, n!)$.

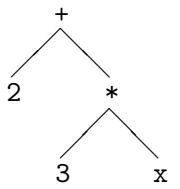
Pour ne pas heurter les habitudes ancestrales, la plupart des langages de programmation utilisent les notations infixées et préfixées ; seuls certains langages spécialisés comme PostScript [1] utilisent une notation postfixée. Tel langage pourra vous autoriser à écrire des expressions mixtes comme $f(x+1)$. Mais s'il vous permettra de définir de nouveaux opérateurs préfixés (les fonctions), il vous interdira sans doute la programmation de nouveaux opérateurs infixés, par exemple un opérateur d'élévation à la puissance $x \wedge y$. Et entre nous il aura bien raison, car le programmeur devrait prévoir l'interaction de ce nouvel opérateur infixé avec tous les autres, et choisir la bonne priorité garantissant l'ordre correct des opérations ! Inutile de vous dire que la tâche n'est pas facile, même pour un professionnel, alors pour le programmeur occasionnel...

Il reste enfin à noter que certains langages ont choisi un codage propre. En Apl [2], l'expression $2 + 3 \times x$ signifie l'addition de 2 et de 3, le résultat étant multiplié par x. Bref, tout n'est pas rose dans le monde des notations.

1. Dérivé du langage Forth, PostScript est un langage destiné aux imprimantes.

2. APL est un langage intéressant mais étrange voué aux mathématiques et à la gestion. Inventé en 1962 par Kenneth Iverson [Ive62], il est tombé en disgrâce à cause de son usage immodéré de lettres grecques, mais nombre de ses idées sont restées vivantes.

À la suite de LISP, notre langage SCHEME a fait le choix radical d'une notation unique, **préfixée**. Autrement dit, la représentation mentale de l'expression $2 + 3 \times x$ sera *l'addition de 2 et du produit de 3 par* x, mettant ainsi l'accent sur ce qui est le plus important dans cette expression, à savoir qu'il s'agit d'une addition. Cette importance de l'opérateur principal d'une formule apparaît clairement dans la représentation de l'expression sous la forme d'un *arbre* portant l'opérateur à la *racine* et dont les opérandes sont les *fils*, eux-mêmes structurés en arbres.



L'intérêt principal de cette homogénéité de notation tient à la grande facilité de les analyser. Cela permet à peu de frais d'écrire de nouveaux prototypes de langages de programmation, ou des programmes symboliques (d'Intelligence Artificielle, de calcul scientifique) dans lesquels les données sont représentées comme des arbres : langage naturel, bases de connaissances, plans d'action, expressions algébriques, graphiques vectoriels, documents XML, compilateurs, etc.

Préfixer une expression ne pose aucun problème tant que le nombre d'opérandes d'un opérateur (on dit aussi son *arité*) est fixé une fois pour toutes. Si l'addition et la multiplication étaient toujours binaires, et le logarithme unaire, l'expression suivante ne porterait aucune ambigüité :

```
+ * x log y + z 1
```

Il est assez clair qu'elle dénote l'addition de $x \log y$ et de $z + 1$. Sachant qu'elle est préfixée, on ne voit pas bien quel autre groupement serait valide. Comment peut-on en retrouver les morceaux ? Tout le problème revient à déterminer les deux points de coupure entre la racine et le fils gauche (le premier opérande), puis entre le fils gauche et le fils droit (le second opérande) :

```
+ | * x log y | + z 1
  <-----------?
```

La première coupure est claire, juste après l'opérateur de tête. La seconde est moins évidente, on n'en connaît pas la longueur à priori : elle peut valoir 1, 3 ou plus, ici elle vaut 4. Il existe un algorithme simple pour trouver la seconde coupure, basée sur le *théorème des poids*. Associons un *poids* de -1 à un opérateur binaire comme `+`, un poids de 0 à un opérateur unaire comme `log`, et un poids de 1 à un nombre. Le théorème en question dit que le poids total d'une expression bien formée est égal à 1, quelle que soit la longueur de l'expression. Mais alors, trouver le second point de coupure est simple. Il suffit de se placer sur le second élément, et d'ajouter les poids successifs rencontrés jusqu'à l'obtention d'un poids total égal à 1. Ce poids 1 obtenu, nous parvenons au point de coupure et une fois la coupure principale obtenue, il suffira par récurrence de relancer la même démarche sur chacun des deux segments trouvés. Il est aussi possible, nous le

programmerons par la suite, de faire tout *en un seul passage* de gauche à droite, i.e. en visitant une seule fois chaque élément. Bref, nous venons de procéder à un reparenthésage de l'expression préfixée non parenthésée :

```
(+ (* x (log y)) (+ z 5))
```

En termes savants, nous avons effectué une *analyse syntaxique* [i.e. grammaticale [3]] d'une phrase d'un langage, le langage des expressions. Ce que nous obtenons est une *expression préfixée complètement parenthésée*. Hélas le théorème des poids ne fonctionne que si les arités des opérateurs sont fixées une fois pour toutes. Or, l'addition ou la multiplication par exemple, étant associatives, peuvent porter sur plusieurs opérandes. Il est quand même plus simple de pouvoir parler de l'addition de x, y et z plutôt que de l'addition de x avec l'addition de y et de z :

`+ x y z` est meilleur que `+ x + y z`

Il faut alors placer systématiquement des parenthèses. L'effort que cela demandera pour entrer les expressions sera plus que largement compensé par la facilité de les analyser. Vous voilà donc prévenus : SCHEME **n'utilise que des expressions préfixées complètement parenthésées** comme `(+ (* 2 3 4) 5)`.

Bien entendu, dans un logiciel réel, de mathématiques ou de langage naturel, rien n'empêchera le programmeur d'écrire un *pré-processeur* d'expressions, un programme les prenant comme l'être humain a été habitué à les aimer, et les transformant comme SCHEME les préfère. Le traitement interne lui, sera fait sur des expressions parenthésées, donc sur des arbres. Si l'on a pu dire en ce sens que SCHEME était un système de traitement d'arbres, cela ne préjuge pas de la forme – linéarisée ou pas – que ces arbres prendront pour des yeux humanoïdes... Mais le fait est qu'avec un peu d'habitude [indispensable], la gêne des parenthèses s'estompe vite et accélère les processus de pensée.

Maths	*Lisp/Scheme*
$f(x, y, z)$	`(f x y z)`
$f(x + 1, y)$	`(f (+ x 1) y)`
$x + f(y)$	`(+ x (f y))`
p et q	`(and p q)`
si $x > 0$ alors $x + 1$ sinon y	`(if (> x 0) (+ x 1) y)`

Remarque — Afin d'empêcher tout de suite un mauvais modèle de se mettre en place, ne croyez pas que dans la mémoire de l'ordinateur, une expression SCHEME contient des parenthèses. Ces denières ne sont qu'un élément de décoration syntaxique pour exprimer la représentation externe d'un arbre ! Les parenthèses ne sont là que pour exprimer dans une écriture unidimensionnelle les bifurcations qui ont lieu dans une arborescence à deux dimensions.

3. Dorénavant, nous utiliserons des crochets à la place des parenthèses dans le texte en français pour ne pas interférer avec les parenthèses du langage de programmation. On ne saurait être trop prudent.

1.3 Le toplevel

1.3.1 Un endroit où l'on cause

Le programmeur SCHEME va rédiger ses programmes à l'aide d'un éditeur de texte. Mais son langage étant traditionnellement interactif, il se trouve naturellement un lieu où il va pouvoir dialoguer avec l'évaluateur SCHEME, que ce dernier soit interprété ou compilé [4]. Cet *endroit où l'on cause* se nomme le **toplevel** [niveau supérieur [5]]. Il est possible d'y entrer une expression au clavier pour obtenir sa valeur. La présence du toplevel est matérialisée par un *prompt* dont la forme va dépendre du système utilisé. Dans ce livre, ce sera le caractère > :

```
> (+ 2 3 4)                    ; ceci est un commentaire (ignoré)
9
> (+ (* 2 3)                   ; une expression peut tenir sur
     (* 3 4)                   ; plusieurs lignes, jusqu'à ce que
     (* 4 5))                  ; la dernière parenthèse soit fermée.
38
```

Si votre expression est syntaxiquement [6] incorrecte, par exemple si vous avez oublié un espace entre + et 2, alors +2 tout attaché sera lu comme le nombre 2, et puisqu'il est vain de vouloir appliquer une pseudo fonction 2, vous obtiendrez alors un message d'erreur plus ou moins explicite sur la nature de la faute :

```
> (+2 3 4)
procedure application: expected procedure, given: 2
```

La *boucle toplevel* est donc un processus éternel [enfin presque...] consistant à :

1. demander à l'utilisateur d'entrer au clavier une suite de caractères formant une expression SCHEME grammaticalement correcte. Cette phase d'*analyse syntaxique* aboutit à la construction en mémoire d'un objet interne $\mathcal{A}$ dont la suite de caractères en question n'est qu'une représentation externe.
2. évaluer l'objet $\mathcal{A}$ ainsi construit pour produire un autre objet $\mathcal{B}$.

4. Cette distinction devient au fil du temps de moins en moins claire, avec l'apparition de systèmes hybrides dont les compilateurs produisent du code virtuel ou *bytecode*, qui sera ensuite mis en correspondance avec les processeurs réels. Ce qui est par contre très important pour le programmeur, et qui n'allait pas du tout de soi avec le LISP traditionnel, c'est que le sens de son programme soit indépendant du fait que le système avec lequel il travaille soit compilé ou interprété. La seule différence visible doit résider dans la vitesse à l'exécution.

5. En DRRACKET, il se trouve dans la portion inférieure de la fenêtre

6. Syntaxique signifie grammatical, tout ce qui concerne la structure, la forme. Cet adjectif s'oppose à *sémantique*, qui parle du sens, de la signification, du fond.

3. afficher une représentation externe de l'objet $\mathcal{B}$ sous la forme d'une suite de caractères.
4. et retourner au point 1...

Il est donc important de faire attention à la distinction entre une *valeur* SCHEME interne et une *représentation externe* de cette valeur, destinée à l'œil humain.

Remarque — DRRACKET maintient l'historique des demandes de calcul au toplevel car il est impossible de modifier les lignes précédentes pour des raisons de cohérence. Par pressions successives de *Esc-p* [previous] ou *Esc-n* [next], on peut faire défiler les demandes antérieures pour en relancer une, en la modifiant au besoin.

L'interactivité du langage n'est pas intrinsèquement liées à la nature de LISP, mais tient aux raisons historiques de son apparition, pour l'intelligence artificielle et le pilotage de systèmes logiciels complexes [*giving it a command, then seeing what happened, then giving it another command...* ainsi que l'expliquait McCarthy], mais aussi à travers l'émergence de recherches sur le partage du temps de calcul dans les systèmes multi-utilisateurs, problème auquel s'attaqueront aussi les concepteurs du langage BASIC, qui a été conçu à l'origine comme un mini-FORTRAN interactif. Les langages interactifs [ou plutôt les implémentations interactives de langages] favorisent le développement incrémental du logiciel, avec beaucoup d'exécutions intermédiaires.

Il ne faudrait pas pour autant réduire le rôle du toplevel à celui d'une calculette de haut niveau. Il permet aussi d'interagir avec le système d'exploitation [Linux, MacOS-X, Windows] pour gérer ses fichiers ou lancer des programmes externes.

1.3.2 Un environnement de variables

Lorsqu'un programmeur parle du *toplevel*, il peut en réalité sous-entendre deux choses. Ou bien le lieu, matérialisé par un prompt, où se déroule l'interaction avec l'évaluateur SCHEME, ou bien un **dictionnaire** de couples <*symbole,valeur*> contenant tous les symboles primitifs, chacun muni de sa valeur initiale. L'un de ces couples pourrait être constitué du symbole `pi` et de sa valeur approchée `#i3.14159265358979`, ou encore du symbole `+` et de sa valeur qui est la procédure d'addition :

```
12 > pi                                   ; que vaut le symbole pi ?
13 #i3.14159265358979                     ; #i ⇔ inexact ⇔ approché
14 > +                                    ; que vaut le symbole + ?
15 #<procedure:+>
```

En effet, l'addition est une fonction mathématique dont le nom en SCHEME est le symbole `+`. La valeur de ce symbole `+` est une *procédure*, un sous-programme en langage-machine qu'il est hors de question d'afficher à l'écran et qui contient le code du processus de calcul. C'est pourquoi l'affichage de la valeur du symbole `+` ressemble à quelque chose

comme `#<procedure:+>`, qui n'est que la représentation externe, pour l'œil humain, d'un véritable objet interne, la procédure d'addition.

Ce dictionnaire du toplevel qui contient les valeurs des symboles primitifs est nommé **dictionnaire global** ou **environnement global** et sera noté $\mathcal{G}$. Il ne contient pas seulement les symboles primitifs, mais aussi tous ceux que le programmeur va y définir. Si l'on tente d'accéder à la valeur d'un symbole non défini, on obtiendra une erreur :

```
16 > foo                                    ; que vaut le symbole foo ?
17 reference to undefined identifier: foo
```

Pour le programmeur, l'activité de programmation consiste à définir de nouvelles fonctions, le langage SCHEME ne pouvant tout prévoir. D'un point de vue naïf, programmer consiste simplement à apprendre des mots nouveaux à un ordinateur.

1.4 Nommer le résultat d'un calcul avec `define`

Le toplevel permet d'effectuer des calculs, mais aussi de donner un nom à un résultat, avec la primitive `define`. Le nom en question sera un *symbole* :

```
18 > pi                                     ; pi a une valeur au toplevel
19 #i3.141592653589793                      ; #i signifie "nombre approché"
20 > (define degre (/ pi 180))              ; la valeur d'un degré en radians
21 > (* 30 degre)                           ; 30 degrés ≃ 0.52 radians
22 #i0.5235987755982988
```

Les **symboles** jouent en SCHEME le rôle de variables ou de mots à l'état brut. Ils sont constitués de caractères sans espace, retour-chariot, tabulation, parenthèses, accolades ou crochets, et sans pour autant bien sûr former un nombre. Voici quelques symboles licites :

```
x  x12  foo  successeur-de  polaire->rect  go!  +  bon?  **
```

Il est d'usage en SCHEME d'utiliser le trait d'union, comme en français : `successeur-de` est plus lisible que `successeur_de` ou `successeurDe`. Les majuscules et minuscules sont en principe différenciées. Le fait de différencier majuscules et minuscules dans les symboles n'est qu'un réglage du niveau de langage *Étudiant avancé*. Nous vous conseillons vivement de cocher la case concernée si ce n'est déjà fait !

Avant de définir un symbole, assurez-vous qu'il n'est pas prédéfini dans le langage. Certains systèmes SCHEME sont assez laxistes pour vous permettre de redéfinir leurs primitives [ce n'est heureusement pas le cas dans notre niveau de langage *Étudiant avancé*]. Ne jouez pas avec le feu : pour vérifier si un symbole possède déjà une valeur, il suffit simplement de le demander.