Une introduction à la programmation en Emacs Lisp

Robert J. Chassell traduit par Laurent Garnier

13 mai 2015

Ceci est une *Introduction à la programmation Emacs Lisp*, pour des gens qui ne sont pas programmeurs.

Edition 3.10, 28 Octobre 2009

Publié par la :

GNU Press,

http://www.fsf.org/campaigns/gnu-press/

une division de la

email : sales@fsf.org

Free Software Foundation, Inc.

Tel : +1 (617) 542–5942

51 Franklin Street, Fifth Floor

Fax : +1 (617) 542–2652

ISBN 1-882114-43-4

Table des matières

Préface

La plupart des environnements de développement intégrés GNU Emacs sont écrits dans le langage de programmation Emacs Lisp. Le code écrit dans ce langage de programmation est le logiciel—l'ensemble des instructions—qui disent à l'ordinateur quoi faire quand vous lui donnez des commandes. Emacs est conçu de sorte que vous pouvez écrire du nouveau code Emacs Lisp et l'installer simplement comme une extension de l'éditeur.

(GNU Emacs est parfois appelé «l'éditeur modifiable», mais il fait bien plus que fournir des possibilités d'édition. C'est mieux de se référer à Emacs comme un «environnement de développement modifiable». Toutefois, cette phrase est un peu pompeuse. C'est plus simple de se référer à Emacs tout simplement comme un éditeur. De plus, tout ce que vous faîtes avec Emacs—trouver une date du calendrier Maya et les phases de la lune, simplifier des polynômes, déboguer du code, gérer des fichiers, lire des lettres, écrire des livres—toutes ces activités sont des façons d'éditer dans le sens le plus général du mot.)

Aussi Emacs Lisp est habituellement pensé seulement en association avec Emacs, c'est un langage de programmation complet.Vous pouvez utiliser Emacs Lisp comme vous le feriez pour n'importe quel autre langage de programmation.

Peut être vous voulez comprendre la programmation ; peut être vous voulez modifier Emacs ; ou peut être vous voulez devenir un programmeur. Cette introduction à Emacs Lisp est conçue pour vous faire démarrer : vous guider dans l'apprentissage des fondamentaux de la programmation, et plus important, vous montrer comment apprendre à aller plus loin.

En lisant ce texte

À travers ce document, vous verrez un petit échantillon de programmes que vous pouvez lancer dans Emacs. Si vous lisez ce document dans Info dans GNU Emacs, vous pouvez lancer les programmes comme ils apparaissent. (C'est facile à faire et c'est expliqué lorsque les programmes sont présentés.) Alternativement, vous pouvez lire cette introduction comme un livre imprimé pendant que vous êtes assis devant un ordinateur faisant tourner Emacs. (C'est ce que j'aime faire ; j'aime les livre imprimés.) Si vous n'avez pas un Emacs qui tourne devant vous, vous pouvez toujours lire ce livre, mais dans ce cas, le mieux c'est de le traiter comme une nouvelle ou un guide de voyage d'un pays pas encore visité : intéressant, mais pas pareil que d'y être.

La plupart de cette introduction est dédiée aux walkthroughs ou tours guidés du code utilisé dans GNU Emacs. Ces tours sont conçus avec deux objectifs : d'abord, vous familiariser avec le réel, le code de travail (le code utilisé tous les jours) ; et, ensuite, vous familiariser avec la façon dont Emacs fonctionne. C'est très intéressant de voir comment un environnement de travail est implémenté. Aussi, j'espère que vous prendrez l'habitude de naviguer dans le code source. Vous pouvez y apprendre et emprunter des idées. Avoir GNU Emacs est comme avoir une cave de dragons pleine de trésors.

En supplément d'en apprendre sur Emacs comme un éditeur et Emacs Lisp comme un langage de programmation, les exemples et les tours guidés vous donneront une opportunité d'avoir de l'acointance avec Emacs comme un environnement de programmation Lisp. GNU Emacs supporte la programmation et fournit des outils que vous voudrez pour devenir à l'aise dans l'usage comme `M-.` (la touche invoque la commande `find-tag`). Vous apprendrez aussi à propos des tampons et autre objets qui font partie de l'environnement. En apprendre au sujet de ces fonctionnalités d'Emacs est comme apprendre les nouvelles routes autour de votre ville.

Finalement, j'espère convey une partie des compétences pour utiliser Emacs et apprendre les aspects de la programmation que vous ne connaissez pas. Vous pouvez utiliser souvent Emacs pour vous aider à comprendre ce qui vous tracasse ou pour trouver comment faire quelque chose de nouveau. Cette auto-reliance n'est pas seulement un plaisir, mais aussi un avantage.

À qui s'adresse ceci

Ce texte est écrit comme une introduction élémentaire pour des gens qui ne sont pas programmeurs. Si vous êtes un programmeur, vous pouvez ne pas être satisfait par ceci. La raison est que vous êtes certainement devenu un expert de la lecture des manuels de référence et exaspéré par la façon dont ce texte est organisé.

Un programmeur expert qui a relu ce texte m'a dit :

Je préfère apprendre avec les manuels de référence. Je «plonge dans» chaque paragraphe et «ressors prendre de l'air» entre les paragraphes.

Quand j'arrive à la fin d'un paragraphe, je suppose que ce sujet est traité, fini, que je sais tout ce dont j'ai besoin (avec la possible exception du cas où le paragraphe suivant démarre en en parlant avec plus de détail). Je m'attends à ce qu'un bon manuel de référence n'aura pas beaucoup de redondance, et qu'il aura d'excellent pointeurs vers l'endroit où trouver l'information que je souhaite.

Cette introduction n'est pas écrite pour cette personne !

D'abord, j'essaie de dire toute chose au moins trois fois : un, pour l'introduire ; deux, pour la montrer dans le contexte ; et trois, pour la montrer dans un contexte différent, ou pour la revoir.

Ensuite, j'ai toujours difficilement mis toute l'information à propos d'un sujet dans un endroit, dans moins qu'un paragraphe. Dans ma façon de penser, cela impose un pavet trop lourd pour le lecteur. À la place j'essaie d'expliquer seulement ce dont vous avez besoin de savoir au moment idoine. (Parfois j'inclus un peu d'information supplémentaire donc vous ne serez pas surpris plus tard quand l'information supplémentaire sera formellement introduite.)

Quand vous lisez ce texte, vous ne vous attendez pas à apprendre tout la première fois. Souvent, vous aurez seulement besoin de faire, comme c'était, un «nodding acquaintance» avec certains des items mentionés. Mon espoir est d'avoir structuré le texte et de vous avoir donné assez d'indice qui vous alerteront sur ce qui est important, et que vous vous concentrerez deçu.

Vous aurez besoin de «plonger dans» certains paragrpahes ; il n'y a pas d'autre façon de les lire. Mais j'ai essayé de réduire le nombre de ces paragraphes. Ce livre est intended comme une colline approchable, plutôt que comme une montagne escarpée.

Cette introduction *Programming in Emacs Lisp* a un document compagnon, *The GNU Emacs Lisp Reference Manual*. Le manuel de référence a beaucoup plus de détails que cette introduction. Dans le manuel de référence, toute l'information à propos d'un sujet est concentrée dans un endroit. Vous devriez y aller si vous êtes comme le programmeur cité plus haut. Et, bien sûr, après avoir lu cette *Introduction*, vous trouverez le *Reference Manual* utile wuand vous écrirez vos propres programmes.

Histoire de Lisp

Lisp a été développé pour la première fois à la fin des années 50 au Massachussetts Institute of Technology pour la recherche en intelligence artificielle. La super puissance du langage Lisp qui le rend supérieur pour d'autres sujets est tant, l'écriture de commandes d'édition et les environnements intégrés.

GNU Emacs Lisp est largement inspiré de Maclisp, qui a été écrit au MIT dans les années 60. C'est en partie ce qui a inspiré Common Lisp, qui est devenu un standard dans les années 80. Toutefois, Emacs Lisp est beaucoup plus simple que Common Lisp. (La distribution standard d'Emacs contient une option d'extension de fichier, `cl.el` qui ajoute beaucoup de fonctionnalités de Common Lisp à Emacs Lisp.)

Remarque pour les novices

Si vous ne connaissez pas GNU Emacs, vous pouvez toujours lire ce document avec profit. Toutefois, je recommande d'apprendre Emacs, au moins d'apprendre comment vous déplacer sur l'écran de votre ordinateur. Vous pouvez apprendre seul comment utilier Emacs avec le tutoriel en ligne. Pour l'utiliser, taper `C-h t`. (Cela signifie pressez et relachez la touche `CTRL` et le `h` en même temps, et puis pressez et relachez `t`.)

Aussi, je ferais souvent référence à l'une des commandes standard d'Emacs en listant les touches que vous presserez pour invoquer la commande et ensuite en donnant le nom de la commande entre parenthèses, comme ça : `M-C-\` (`indent-region`). Ce qui signifie que la commande `indent-region` est personnellement invoquée en tapant `M-C-\`. (Vous pouvez, si vous souhaitez, changer les touches qui sont tapées pour invoquer la commande ; cela est appelé *redéfinir*. Voir section 16.8 «Keymaps», page 57.) L'abréviation `M-C-\` signifie que vous tapez votre touche `META`, touche `CTRL` end touche `\` toutes en même temps. (Sur de nombreux claviers modernes la touche `META` est étiquetée `ALT`.) Parfois une combinaison comme celle-là est appelée un accord de touche, de façon similaire à un accord de piano. Si votre clavier n'a pas de touche `META`, la touche `ESC` préfixée est utilisée à la place. Dans ce cas, `M-C-\` signifie que vous pressez puis relachez la touche `ESC` et après tapez la touche `CTRL` et la touche `\` en même temps. Mais usuellement `M-C-\` signifie pressez la touche `CTRL` along avec la touche nommée `ALT` et, en même temps, pressez la touche `\`.

De plus taper a lone accord de touche, vous pouvez préfixer ce que vous tapez avec `C-u`, qui est appelé «l'argument universel». L'accord de touche `C-u` passe un argument à la commande subsequent. Alors, pour indenter une région de texte plein par 6 espaces, marquez la région, et ensuite tapez `C-u 6 M-C\`. (Si vous ne spécifiez pas un nombre, Emacs passera le nombre 4 à la commande ou alors lancera la commande différemment que ce que vous souhaitiez.) Voir section «arguments numériques» dans *The GNU Emacs Manual*.

Si vous êtes en train de lire ça dans Info utilisant GNU Emacs, vous pouvez lire à travers ça tout le document juste en pressant la barre espace, `SPC`. (Pour en apprendre sur Info tapez `C-h i` et ensuite sélectionnez Info.)

Une remarque sur la terminologie : quand j'utilise le mot Lisp tout seul, je fais souvent référence aux variantes des dialectes de Lisp en général, mais quand je parle de Emacs Lisp, je fais référence à GNU Emacs Lisp en particulier.

Remerciements

Mes remerciements à tous ceux qui m'ont aidé pour ce livre. Un remerciement spécial à Jim Blandy, Noah Friedman, Jim Kingdon, Roland McGrath, Frank Ritter, Randy Smith, Richard M. Stallman, et Melissa Weisshaus. Mes remerciements également à Philip Johnson et David Stampe pour leur encouragement. Mes erreurs sont miennes.

Robert J. Chassell
bob@gnu.org

Chapitre 1

Traitement de liste

Pour un œil non exercé, Lisp est un langage de programmation étrange. Dans du code Lisp il y a des parenthèses partout. Certains prétendent même que le nom signifie «Lots of Isolated Silly Parentheses[1]». Mais cette prétention est injustifiée. Lisp signifie LISt Processing[2], et le langage de programmation gère les listes (et listes de listes) en les mettant entre parenthèses. Les parenthèses marquent les limites de la liste. Parfois, une liste est précédée par une apostrophe ou guillemet, «'»[3] les listes sont la base de Lisp.

1.1 Listes Lisp

En Lisp, une liste ressemble à ça : `'(rose violet daisy buttercup)`. Cette liste est précédée d'une apostrophe. Elle pourrait être écrit comme suit, ce qui ressemble plus au type de liste dont vous avez l'habitude :

```
'(rose
violet
daisy
buttercup)
```

Les éléments de cette liste sont les noms de quatre fleurs différentes, séparés par des espaces et entourés par des parenthèses, comme des fleurs dans un champ avec un mur de pierre autour d'elles.

Les listes peuvent aussi contenir des nombres, comme dans cette liste : `(+ 2 2)`. Cette liste a un signe plus, «`+`», suivi par deux «`2`», chacun séparé par un espace.

En Lisp, les données et les programmes sont représentés de la même façon ; autrement dit, ils sont tous les deux des listes de mots, de chiffres ou d'autres listes, séparées par des espaces et entourées par des parenthèses. (Puisqu'un programme ressemble à des données, un programme peut facilement servir de donnée pour un autre, ce qui est une fonctionnalité très puissante de Lisp.) (Soit dit en passant, ces deux remarques entre parenthèses ne sont pas des listes Lisp, car elles contiennent des « `;` » et «`.`» comme signes de ponctuation.)

Voici une autre liste, cette fois avec une liste à l'intérieur :

```
'(this list has (a list inside of it))
```

[1] Beaucoup de parenthèses idiotes isolées

[2] traitement de liste

[3] L'apostrophe ou guillemet simple est l'abréviation de la fonction citation ; les fonctions sont définies dans la section 1.3 Générer un message d'erreur, page 4

Les composants de cette liste sont les mots «`this`», «`list`», «`has`», et la liste «`(a list inside of it)`». La liste intérieure est constituée des mots «`a`», «`list`», «`inside`», «`of`», «`it`».

1.1.1 Atomes Lisp

En Lisp, ce que nous appelons mots est appelé *atomes*. Ce terme vient du sens historique du mot atome, qui signifie «indivisible». En ce qui concerne Lisp, les mots que nous avons utilisés dans les listes ne peuvent être divisés en parties plus petites et de même dans le cadre d'un programme ; même avec des chiffres et des symboles de caractères unique comme «`+`». D'autre part, contrairement à un atome antique, une liste peut être divisée en plusieurs parties.(Voir chapitre 7 «`car cdr & cons` Fonctions Fondamentales», page 39.)

Dans une liste, les atomes sont séparés les uns des autres par des espaces. Ils peuvent être juste à côté d'une parenthèse.

Techniquement parlant, une liste en Lisp consiste en une paire de parenthèses entourant des atomes séparés par des espaces ou entourant d'autres listes ou entourant à la fois des atomes et d'autres listes. Une liste peut n'avoir qu'un seul atome ou ne contenir rien du tout. Une liste avec rien du tout ressemble à ça : (), et est appelée la *liste vide*. Contrairement à n'importe quoi d'autre, une liste vide est considérée à la fois comme un atome et comme une liste.

La représentation imprimée des atomes et listes est appelée *symbolic expressions* ou, de façon plus concise, *s-expressions*. Le mot *expression* par lui-même peut référer à la fois à la représentation imprimée, ou à la liste comme elle est perçue dans l'ordinateur. Souvent, les gens utilisent le terme *expression* sans distinction. (Aussi, dans beaucoup de textes, le mot *forme* est utilisé comme synonyme pour expression.)

Incidemment, les atomes qui contituent notre univers étaient nommés de telle façon lorsque nous pensions qu'ils étaient indivisibles. Les parties peuvent s'éclater en atome ou peuvent se fissionner en deux parties à peu près égales en taille. Les atomes physique ont été nommé prématurément, avant que leur vraie nature ne soit découverte. En Lisp, certains types d'atome, comme un tableau, peuvent être séparés en plusieurs parties ; mais le mécanisme pour faire cela est différent de celui pour éclater une liste. Autant qu'une liste d'opération est concernée, les atomes d'une liste sont irréductible.

Comme en Anglais, les sens de la composition des lettres d'un atome Lisp sont différents du sens que les lettres font pour former un mot. Par exemple, le mot pour le sud américain sloth, le '`ai`', est complètement différent des deux mots, '`a`', et '`i`'.

Il y a beaucoup de types d'atome dans la nature mais seulement quelques uns en Lisp : par exemple, *numbers*, comme 37, 511, ou 1729, et *symbols*, comme '`+`', '`foo`', ou '`forward-line`'. Les mots que nous avons listés dans les exemples au-dessus sont tous des symboles. Dans une conversation Lisp de tous les jours, le mot “atome” n'est pas souvent utilisé, parce que les programmeurs essaient d'habitude d'être plus spécifique au sujet du type d'atome avec lesquels ils travaillent. La programmation Lisp est beaucoup plus à propos des symboles (et parfois des nombres) avec des lists. (Incidemment, les trois mots précédant la remarque entre parenthèse forment un liste en Lisp, dès qu'elle contient des atomes, qui sont dans ce cas des symboles, séparés par des espaces et encapsulés par des parenthèses, sans acune ponctuation non-Lips.)

Le texte entre les guillemets—même des phrases ou paragraphes—est aussi un atome. Voici un exemple :

```
'(this list includes ``text between quotation marks.'')
```

En Lisp, tous les textes entre les guillemets incluant la marque de ponctuation et les espaces est un atome simple. Ce type d'atome est appelé une *chaîne* (pour 'chaîne de caractères') et est le type de chose qui est utilisée pour des messages que l'ordinateur peut afficher pour un

humain. Les chaînes sont un type d'atome différent des nombres ou des symboles et sont utilisées différemment.

1.1.2 Espaces dans les listes

Le nombre d'espace dans une liste ne compte pas. Du point de vue du langage Lisp,

```
'(this list
  looks like this)
```

est exactement identique à ça :

```
'(this list looks like this)
```

Les deux exemples montrent ce qui pour Lisp est la même liste, la liste composée des symboles '`this`', '`list`', '`looks`', '`like`', et '`this`' dans cet ordre.

Les espaces supplémentaires ou les nouvelles lignes sont faites pour construire une liste plus lisible pour les humains. Quand Lisp lit les expressions, il se débarrasse de tous les espaces blancs supplémentaires (mais il doit avoir au moins un espace entre les atomes afin de les distinguer.)

Aussi étrange que cela semble, les exemples que nous avons vu couvrent la quasi-totalité des listes en Lisp ! Toute autre liste en Lisp ressemble plus ou moins à l'un de ces exemples, sauf que la liste pourrait être plus longue et plus complexe. En bref, une liste est entre parenthèses, une chaîne est entre guillemets, un symbole ressemble à un mot, et un nombre ressemble à un nombre. (Pour certaines situations, des crochets, des points et quelques autres caractères spéciaux peuvent être utilisés ; cependant, nous allons aller très loin sans eux.)

1.1.3 GNU Emacs vous aide à taper les listes

Lorsque vous tapez une expression Lisp dans GNU Emacs en utilisant soit le mode d'interaction Lisp ou le mode Emacs Lisp, vous avez à votre disposition plusieurs commandes pour formater l'expression Lisp de sorte qu'il soit facile à lire. Par exemple, appuyer sur la touche `TAB` indente automatiquement la ligne où se trouve le curseur avec le bon décalage. Une commande pour indenter proprement le code dans une zone est habituellement liée à `M-C \-`. L'indentation est conçue de sorte que vous pouvez voir quels sont les éléments d'une liste qui appartiennent à la liste—les éléments d'une sous-liste sont plus indentés que les éléments de la liste qui l'encapsule.

En outre, lorsque vous tapez une parenthèse fermante, Emacs saute momentanément le curseur sur la parenthèse ouvrante correspondante, de sorte que vous pouvez voir laquelle est-ce. C'est très utile, car chaque liste que vous tapez dans Lisp doit avoir sa parenthèse fermante correspondant à celle ouvrante. (Voir la section «mode majeurs» dans le Manuel GNU Emacs, pour plus d'information sur les modes Emacs.)

1.2 Lancer un programme

Une liste en Lisp—toute—liste est un programme prêt à fonctionner. Si vous l'exécutez (en jargon Lisp on dit évaluer), l'ordinateur fera l'une des trois choses : ne rien faire, sauf renvoyer la liste elle-même ; vous envoyer un message d'erreur ; ou traiter le premier symbole dans la liste comme une commande pour faire quelque chose. (Habituellement, bien sûr, c'est le dernier cas que vous voulez vraiment !)

L'apostrophe (ou guillemet simple), ', que j'ai mise en face de quelques-uns des exemples de listes dans les sections précédentes est appelé quote[4] ; quand il précède une liste, il dit à Lisp de ne rien faire avec la liste, autre que de la prendre comme elle est écrite. Mais s'il n'y a pas de quote précédent une liste, le premier élément de la liste est spécial : c'est une commande à laquelle l'ordinateur doit obéir. (En Lisp, ces commandes sont appelées fonctions.) La liste `(+ 2 2)` ci-dessus n'a pas de quote la précédant, de sorte que Lisp comprend que le `+` est une instruction pour faire quelque chose avec le reste de la liste : ajouter les chiffres qui suivent.

Si vous lisez ce manuel à depuis GNU Emacs dans Info, voici comment vous pouvez évaluer une telle liste : placez votre curseur immédiatement après la parenthèse de droite de la liste suivante et puis tapez `C-x C-e` :

```
(+ 2 2)
```

Vous verrez que le nombre 4 appraît dans la zone écho. (Dans le jargon, ce que vous venez de faire est "d'évaluer la liste." La zone d'écho est la ligne en bas de l'écran qui affiche (ou "fait échos" au) le texte.) Maintenant, essayez la même chose avec une liste avec quote : placer le curseur après la liste suivante et tapez `C-x C-e` :

```
'(this is a quoted list)
```

Vous verrez `(this is a quoted list)` apparaître dans la zone d'écho.

Dans les deux cas, ce que vous faites est de donner un ordre à l'intérieur du programme GNU Emacs appelé l'interprète Lisp—lui donnant une commande pour évaluer l'expression Lisp. Le nom de l'interprète Lisp vient du mot pour la tâche accomplie par un être humain qui traduit une langue en une autre.

Vous pouvez également évaluer un atome qui ne fait pas partie d'une liste et un qui n'est pas entouré par des parenthèses ; encore une fois, l'interprète Lisp traduit de l'expression lisible par l'homme vers la langue de l'ordinateur. Mais avant de discuter de cela (voir section 1.7 "Variables", page 8), nous allons discuter de ce que l'interprète Lisp fait quand vous faites une erreur.

1.3 Générer un message d'erreur

Partant de sorte que vous ne vous inquiétez pas si vous le faites accidentellement, nous allons maintenant donner un ordre à l'interprète Lisp qui génère un message d'erreur. C'est une activité inoffensive ; et en effet, nous allons souvent essayer de générer des messages d'erreur intentionnellement. Une fois que vous comprenez le jargon, les messages d'erreur peuvent être informatif. Au lieu d'être appelé "messages d'erreur", ils devraient être appelés messages "d'aide". Ils sont comme des panneaux pour un voyageur dans un pays étranger ; les déchiffrer peut être difficile, mais une fois compris, ils peuvent indiquer la voie.

Le message d'erreur est généré par un débogueur GNU Emacs intégré. Nous allons «entrer dans le débogueur». Pour sortir du débogueur taper `q`.

Ce que nous allons faire c'est d'évaluer une liste qui n'a pas de quote la précédant et qui n'a pas de commande significative comme premier élément. Voici une liste presque exactement la même que celle que nous avons utilisé, mais sans la quote la précédant. Placez le curseur juste après et taper `C-x C-e` :

```
(this is an unquoted list)
```

[4] En anglais dans le texte car la traduction «devis» ne m'a pas paru pertinente. De plus quote est bien plus court à écrire que guillemet simple ou apostrophe d'autant qu'il ne s'agit pas ici d'un symbole de ponctuation mais d'un symbole propre au langage Lisp.

(this is an unquoted list) Une fenêtre `*Backtrace*` s'ouvrira et vous devriez voir ce qui suit :

```
Debugger entered--Lisp error : (void-function this)
(this is an unquoted list)
eval((this is an unquoted list))
eval-last-sexp-1(nil)
eval-last-sexp(nil)
call-interactively(eval-last-sexp)
```

Votre curseur sera dans cette fenêtre (vous pouvez avoir à attendre quelques secondes avant qu'il ne devienne visible). Pour quitter le débogueur et faire la fenêtre du débogueur s'en aller, tapez : `q`.

S'il vous plaît tapez `q` maintenant, alors vous devenez plus confiant et vous pouvez sortir du débogueur. Ensuite, tapez `C-x C-e` de nouveau pour y rentrer.

Sur la base de ce que nous savons déjà, nous pouvons presque lire ce message d'erreur.

Vous avez lu le tampon `*Backtrace*` de bas en haut ; il vous dit ce que Emacs fait. Lorsque vous avez tapé `C-x C-e`, vous avez fait un appel à la commande `eval-last-sexp`. `eval` est une abréviation pour «évaluer» et `sexp` est une abréviation pour «expression symbolique». La signification de cette commande est «évaluer la dernière expression symbolique», qui est l'expression juste avant le curseur.

Chaque ligne ci-dessus vous indique ce que l'interprète Lisp évalue après. L'action la plus récente est en haut. Le tampon est appelé `*Backtrace*` car il vous permet de suivre Emacs en arrière.

En haut du tampon `*Backtrace*`, vous voyez la ligne :

```
Debugger entered--Lisp error : (void-function this)
```

L'interprète Lisp a tenté d'évaluer le premier atome de la liste, le mot «this». C'est cette action qui a généré le message d'erreur «void-finction this».

Le message contient les mots «void-function» et «this». Le mot «function» a été mentionné une seule fois auparavant. C'est un mot très important.

Pour nos fins, nous pouvons la définir en disant qu'une fonction est un ensemble d'instructions données à l'ordinateur qui signale à l'oridnateur de faire quelque chose.

Maintenant, nous pouvons commencer à comprendre le message d'erreur : «`void-function this`». La fonction (ce mot est le mot «`this`») n'a pas de définition d'un ensemble d'instructions que l'ordinateur peut mener à bien.

Le mot un peu bizarre, «`void-function`», est conçu pour couvrir la façon dont Emacs Lisp est mis en œuvre, qui est que lorsqu'un symbole n'a pas de définition de fonction attaché à lui, la place qui doit contenir les instructions est «`void`».

D'autre part, puisque nous avons pu ajouter 2 plus 2 avec succès, en évaluant `(+ 2 2)` nous pouvons en déduire que le symbole `+` doit avoir un ensemble d'instructions pour l'ordinateur qui doit obéir et ces instructions doivent être d'ajouter les nombres qui suivent le `+`.

Il est possible de prévenir Emacs d'entrer dans le débogueur dans de tels cas. Nous n'expliquons pas comment faire ici, mais nous allons parler de ce à quoi le résultat ressemble, parce que vous pouvez rencontrer une situation similaire s'il y a un bogue dans un code Emacs que vous utilisez. Dans de tels cas, vous verrez seulement une ligne de message d'erreur ; il apparaîtra dans la zone écho et ressemblera à ceci :

```
Symbol's function definition is void : this
```

Le message disparaît dès que vous tapez une touche, même juste pour déplacer le curseur. Nous connaissons le sens du mot «`Symbol`». Il se réfère au premier atome de la liste, le mot «`this`». Le mot «`function`» se réfère aux instructions qui indiquent à l'ordinateur ce qu'il faut faire. (Techniquement, le symbole indique à l'ordinateur où trouver les instructions, mais c'est une complication que nous pouvons ignorer pour le moment.) Le message d'erreur peut être compris : «`Symbol's function definition is void : this`». Le symbole (c'est le mot «`this`») manque d'instructions pour que l'ordinateur les mènent à bien.

1.4 Noms de symboles et noms de fonctions

Nous pouvons exprimer une caractéristique de Lisp basée sur ce que nous avons discuté jusqu'à présent—une caractéristique importante : un symbole, comme `+`, n'est pas lui-même l'ensemble des instructions de l'ordinateur nécessaire pour mener à bien sa fonction. Au contraire, le symbole est utilisé, peut-être temporairement, comme un moyen de localisation de la définition ou un ensemble d'instructions. Ce que nous voyons est le nom par lequel les instructions peuvent être trouvées. Les noms de personnes fonctionnent de la même manière. Je peux être «`Bob`» ; cependant, je ne suis pas les lettres «`B`», «`o`», «`b`», mais je suis ou étais, la conscience toujours associée à une forme de vie particulière. Le nom n'est pas moi, mais il peut être utilisé pour se référer à moi.

En Lisp, un ensemble d'instructions peut être attaché à plusieurs noms. Par exemple, les instructions informatiques pour l'ajout de nombres peuvent être liées au symbole, plus ainsi que pour le symbole `+` (et sont dans certains dialectes de Lisp). Chez les humains, je peux être appelé «`Robert`» ainsi que «`Bob`» et par d'autres mots ainsi.

D'autre part, un symbole peut avoir seulement une définition de la fonction attachée à elle à la fois. Sinon, l'ordinateur serait confus quant à la définition à utiliser. Si c'était le cas chez les personnes, une seule personne dans le monde pourrait être nommé «`Bob`». Cependant, la définition de la fonction à laquelle le nom se réfère peut être modifié facilement. (Voir la section 3.2 «Installer une définition de fonction», page 27.)

Depuis Emacs Lisp est grand, il est d'usage de symboles nom d'une manière qui identifie la partie d'Emacs auquel la fonction appartient. Ainsi, tous les noms de fonctions qui traitent de Texinfo commencent par «`texinfo-`» et ceux pour les fonctions qui traitent de la lecture du courrier de départ avec «`rmail-`».

1.5 Interpréteur Lisp

Sur la base de ce que nous avons vu, nous pouvons maintenant commencer à comprendre ce que l'interprète Lisp fait lorsque nous commandons à évaluer une liste. Tout d'abord, il semble pour voir s'il y a une quote avant que la liste ; s'il y a, l'interprète nous donne juste la liste. D'un autre côté, s'il n'y a pas de quote, l'interprète regarde le premier élément de la liste et voit si elle a une définition de fonction. Si c'est le cas, l'interpréteur exécute les instructions dans la définition de fonction. Sinon, l'interpréteur affiche un message d'erreur.

C'est ainsi que fonctionne Lisp simple. Il y a des complications supplémentaires que nous allons obtenir dans une minute, mais ce sont les fondamentaux. Bien sûr, pour écrire des programmes Lisp, vous devez savoir comment écrire les définitions de fonctions et leur donner des noms, et comment le faire sans confondre vous-même ou l'ordinateur.

Maintenant, pour la première complication, en plus de la liste, l'interpréteur Lisp peut évaluer un symbole qui n'a pas de quote et qui n'a pas de parenthèses autour de lui. L'interprète Lisp

tentera de déterminer la valeur du symbole comme une variable. Cette situation est décrite dans la section sur les variables. (Voir la section 1.7 «Variables», page 8.)

La deuxième complication se produit parce que certaines fonctions sont inhabituelles et ne fonctionnent pas de manière habituelle. Celles qui ne le font pas sont appelées formes spéciales. Elles sont utilisées pour des tâches spéciales, comme la définition d'une fonction, et il n'y en a pas beaucoup. Dans les prochains chapitres, vous serez initié à plusieurs de ces formes spéciales les plus importantes.

La troisième et dernière complication est la suivante : si la fonction que l'interprète Lisp regarde n'est pas une forme particulière, et si elle fait partie d'une liste, l'interprète Lisp regarde pour voir si la liste a une liste à l'intérieur de celui-ci. S'il y a une liste interne, l'interprète Lisp décide ce qu'il doit faire avec la liste intérieur en premier, puis il travaille sur la liste extérieure. S'il y a encore une autre liste incorporée à l'intérieur de la liste intérieure, cela fonctionne sur celui-là en premier, et ainsi de suite. Il travaille toujours sur la liste la plus profonde en premier. L'interprète travaille sur la liste la plus interne d'abord, pour évaluer le résultat de cette liste. Le résultat peut être utilisé par l'expression l'encapsulant.

Sinon, l'interprète travaille de gauche à droite, d'une expression à l'autre.

1.5.1 Compilation d'octet

Un autre aspect de l'interprétation : l'interprète Lisp est capable d'interpréter deux types d'entités : du code lisible par des humains, sur lequel nous nous concentrerons exclusivement, et du code spécialement traité, appelé byte code compilé, qui n'est pas lisible par les humains. Le byte code compilé s'exécute plus rapidement que le code lisible par les humains.

Vous pouvez transformer un code lisible pour les humains en un byte code compilé en exécutant l'une des commandes de compilation telles que `byte-compile-file`. Le byte code compilé est généralement stocké dans un fichier qui se termine par une extension `.elc` plutôt qu'une extension `.el`. Vous verrez deux types de fichier dans le répertoire `emacs/lisp` ; les fichiers à lire sont ceux qui portent l'extension `.el`.

En pratique, pour la plupart des choses que vous pourriez faire pour personnaliser ou étendre Emacs, vous n'avez pas besoin d'une compilation de byte code ; et je ne vais pas en parler ici. Voir la section «Byte compilation» dans le manuel Emacs Lisp reference, pour une description complète de la byte compilation.

1.6 Évaluation

Lorsque l'interpréteur Lisp fonctionne sur une expression, le terme de l'activité est appelée évaluation. Nous disons que l'interprète «évalue l'expression». J'ai utilisé ce terme plusieurs fois avant. Le mot vient de son utilisation dans le langage courant, «pour déterminer la valeur ou la quantité de ; d'évaluer», selon le Webster New Collegiate Dictionnary.

Après avoir évalué une expression, l'interprète Lisp va probablement renvoyer la valeur que l'ordinateur produit en effectuant les instructions trouvées dans la définition de fonction, ou peut-être qu'il donnera pour cette fonction un message d'erreur. (L'interprète peut également se trouver jeté, pour ainsi dire, à une fonction différente ou il peut tenter de répéter sans cesse ce qu'il fait à tout jamais dans ce qu'on appelle une «boucle infinie». Ces actions sont moins fréquentes ; et nous pouvons les ignorer.) Le plus souvent, l'interprète renvoie une valeur.

Dans le même temps l'interprète renvoie une valeur, il peut faire autre chose ainsi, comme déplacer un curseur ou copeir un fichier ; cet autre type d'action est appelé un effet secondaire. Les actions que nous les humains pensont être importantes, telles que les résultats d'impression,

sont souvent des «effets secondaires» à l'interprète Lisp. Le jargon peut paraître étrange, mais il s'avère qu'il est assez facile d'apprendre à utiliser des effets secondaires.

En résumé, l'évaluation d'une expression symbolique incite le plus souvent l'interprète Lisp à renvoyer une valeur et peut être à réaliser un effet secondaire ; ou bien produire une erreur.

1.6.1 Évaluation des listes internes

Si l'évaluation s'applique à une liste qui est à l'intérieur d'une autre liste, la liste extérieure peut utiliser la valeur renvoyée par la première évaluation comme information lorsque la liste extérieure sera évaluée. Cela explique pourquoi les expressions intérieures sont évaluées en premier : les valeurs qu'elles renvoient sont utilisées par les expressions extérieures.

Nous pouvons enquêter sur ce processus en évaluant un autre exemple de plus. Placez votre curseur après l'expression suivante et tapez `C-x C-e` :

```
(+ 2 (+ 3 3))
```

Le nombre 8 apparaîtra dans la zone d'écho.

Ce qui se passe c'est que l'interprète Lisp évalue d'abord l'expression intérieure, `(+ 3 3)`, pour laquelle la valeur renvoyée est 6 ; il évalue l'expression externe comme s'il était écrit `(+ 2 6)`, qui renvoie la valeur 8. Comme il n'y a pas d'expressions plus englobantes pour évaluer, l'interprète affiche cette valeur dans la zone d'écho.

Maintenant, il est facile de comprendre le nom de la commande invoquée par le raccourci `C-x C-e` : le nom est `eval-last-sexp`. Les lettres `sexp` sont une abréviation pour «expression symbolique», et `eval` pour «évaluer». La commande signifie «évaluer la dernière expression symbolique».

À titre expérimental, on peut essayer d'évaluer l'expression en plaçant le curseur au début de la ligne suivante immédiatement après l'expression, ou l'expression à l'intérieur.

Voici une autre copie de l'expression :

```
(+ 2 (+ 3 3))
```

Si vous placez le cursuer au début de la ligne blanche et tapez `C-x C-e`, vous obtiendrez toujours la valeur 8 imprimée dans la zone d'écho. Maintenant, essayez de mettre le curseur dans l'expression. Si vous le mettez juste après l'avant-dernière parenthèse (il apparaît donc de s'asseoir sur le dessus de la dernière parenthèse), vous obtiendrez un 6 imprimé dans la zone écho ! C'est parce que la commande évalue l'expression `(+ 3 3)`.

Maintenant, mettez le curseur immédiatement après un certain nombre. Type `C-x C-e` et vous obtiendrez le nombre lui-même. En Lisp, si vous évaluez un certain nombre, vous obtenez le nombre lui-même, c'est en cela que les nombres diffèrent des symboles. Si vous évaluez une liste à partir d'un symbole comme `+`, vous aurez une valeur renvoyée c'est le résultat de l'ordinateur après exécution des instructions de la définition de la fonction associée à ce nom. Si un symbole en lui-même est évalué, quelque chose de différent se produit, comme nous le verrons dans la prochaine section.

1.7 Variables

Dans Emacs Lisp, un symbole peut avoir une valeur associée à lui tout comme il peut avoir une définition de fonction associée. Les deux sont différents. La définition de la fonction est un ensemble d'instructions auquelles un ordinateur obéira. Une valeur, d'autre part, est quelque chose, comme un nombre ou un nom, qui peut varier (c'est pourquoi un tel symbole est appelé

variable). La valeur d'un symbole peut être n'importe quelle expression Lisp, comme un symbole, un nombre, une liste, ou une chaîne. Un symbole qui a une valeur est souvent appelé une variable.

Un symbole peut avoir à la fois une définition de fonction et une valeur fixée à lui en même temps. Ou il peut avoir l'un ou l'autre. Les deux sont séparés. C'est un peu similaire à la façon dont le nom Cambridge peut référer à la ville du Massachussets et avoir quelques informations attachées au nom ainsi, comme «grand centre de programmation».

Une autre façon de penser à ce sujet est d'imaginer un symbole comme étant une commode. La définition de la fonction est mise dans un tiroir, la valeur dans un autre et ainsi de suite. Ce qui est mis dans le tiroir contenant la valeur peut être modifié sans affecter le contenu du tiroir maintenant la définition de la fonciton, et vice-versa.

La variable `fill-column` illustre un symbole avec une valeur attachée à elle : dans chaque tampon GNU Emacs, ce symbole est réglé à une valeur, généralement 72 ou 70, mais parfois à une autre valeur. Pour trouver la valeur de ce symbole, d'évaluer par lui-même. Si vous lisez ceci dans Info à l'intérieur de GNU Emacs, vous pouvez le faire en le curseur après le symbole et en tapant `C-x C-e` :

```
fill-column
```

Après que j'ai tapé `C-x C-e`, Emacs a imprimé le nombre 70 dans ma zone d'écho. c'est la valeur pour laquelle `fill-column` est réglée pour moi qui écris ceci. Il peut être différent pour vous dans votre tampon Info. Notez que la valeur renvoyée comme une variable est imprimée exactement de la même manière que la valeur renvoyée par la fonction d'exécution des instructions. Du point de vue de l'interprète Lisp, une valeur renvoyée est une valeur renvoyée. Peu importe le genre d'expression une fois que la valeur est connue.

Un symbole peut avoir n'importe quelle valeur attachée à lui ou, pour utiliser le jargon, on peut lier la variable à une valeur : à un certain nombre, comme `72` ; à une chaîne, «`telle que ça`», à une liste, comme `(spruce pine oak)` ; nous pouvons même lier une variable à une définition de fonction.

Un symbole peut être lié à une valeur de plusieurs façons. Voir la section 1.9 «Réglage de la valeur d'une variable», page 14, pour des informations sur une façon de le faire.

1.7.1 Message d'erreur pour un symbole sans fonction

Lorsque nous avons évalué `fill-column` pour trouver sa valeur en tant que variable, nous n'avons pas placé des parenthèses autour du mot. C'est parce que nous n'avons pas l'intention de l'utiliser comme un nom de fonction.

Si `fill-column` était le premier ou le seul élément d'une liste, l'interprète Lisp tenterait de trouver la définition de fonction attachée à elle. Mais `fill-column` n'a pas de définition de fonction. Essayez d'évaluer ceci :

```
(fill-column)
```

Vous allez créé un tampon `*Backtrace*` disant ceci :

```
Debugger entered--Lisp error : (void-function fill-column)
(fill-column)
eval((fill-column))
eval-last-sexp-1(nil)
eval-last-sexp(nil)
call-interactively(eval-last-sexp)
```

(Rappelez-vous, pour quitter le débogueur et faire la fenêtre du débogueur s'en aller, tapez `q` dans le tampon `*Backtrace*`.)

1.7.2 Message d'erreur pour un symbole sans valeur

Si vous essayez d'évaluer un symbole qui ne possède pas une valeur qui lui est liée, vous recevrez un message d'erreur. Vous pouvez voir cela en expérimentant avec notre addition 2 plus 2. Dans l'expression suivante, placez votre curseur à droite après le `+`, avant le premier nombre 2, tapez `C-x C-e` :

```
(+ 2 2)
```

Vous allez créé un tampon `*Backtrace*` disant ceci :

```
Debugger entered--Lisp error : (void-variable +)
eval(+)
eval-last-sexp-1(nil)
eval-last-sexp(nil)
call-interactively(eval-last-sexp)
```

(Encore une fois, vous pouvez quitter le débogueur en tapant `q` dans le tampon `*Backtrace*`.)

Ce backtrace est différent du premier message d'erreur que nous avons vu, qui dit «`Debugger entered--Lisp error : (void-function this)`». Dans ce cas, la fonction n'a pas de valeur en tant que variable ; tandis que dans l'autre message d'erreur, la fonction (le mot «`this`») n'avait pas de définition.

Dans cette expérience avec le `+`, ce que nous faisions était provoquer l'interprète Lisp évaluer le `+` et rechercher la valeur de la variable au lieu de la définition de la fonction. Nous l'avons fait en plaçant le curseur juste après le symbole plutôt qu'après la parenthèse de la liste englobante que nous avons fait avant. En conséquence, l'interprète Lisp a évalué la s-expression précédente, qui dans ce cas était `+` lui-même.

Depuis `+` n'a pas de valeur liée à elle, juste la définition de fonction, le message d'erreur rapportait que la valeur du symbole comme variable était nulle.

1.8 Arguments

Pour voir comment l'information est transmise à des fonctions, regardons de nouveau notre vieille veille, l'ajout de deux plus deux. En Lisp, ça s'écrit comme suit :

```
(+ 2 2)
```

Si vous évaluez cette expression, le nombre 4 apparaîtra dans votre zone écho. Ce que fait l'interprète Lisp c'est d'ajouter les nombres qui suivent le `+`.

Les nombres ajoutés par `+` sont appelés les arguments de la fonction `+`. Ces nombres sont les informations que l'on donne à ou transmis à la fonction.

Le mot «argument» vient de la façon dont il est utilisé en mathématiques et ne se réfère pas à une dispute entre deux personnes ; au contraire, il se réfère à l'information présentée à la fonction, dans ce cas, à la fonction `+`. En Lisp, les arguments d'une fonction sont les atomes ou les listes qui suivent la fonction. Les valeurs renvoyées par l'évaluation de ces atomes ou des listes sont passées à la fonction. Différentes fonctions nécessitent différents nombres d'arguments ; certaines fonctions n'en exigent pas du tout[5].

[5] Il est curieux de suivre le chemin par lequel le mot «argument» est venu d'avoir deux significations différentes, l'une en mathématiques et l'autre en anglais de tous les jours. Selon le Oxford English Dictionary, le mot dérive du Latin «`faire comprendre, prouver`» ; ainsi venu à signifier, par un fil de dérivation, «la preuve présentée comme preuve», c'est-à-dire, «l'information offerte», qui a conduit à sa signification en Lisp. Mais dans l'autre fil

1.8.1 Arguments des types de données

Le type de données qui doivent être transmises à une fonction dépend de quel type d'information elle utilise. Les arguments d'une fonction telle que `+` doivent avoir des valeurs qui sont des nombres, depuis `+` ajoute des nombres. D'autres fonctions utilisent différents types de données pour leurs arguments.

Par exemple, la fonction `concat` relie ou réunit deux ou plusieurs chaînes de caractères pour produire une chaîne. Les arguments sont des chaînes. La concaténation des deux chaînes de caractères `abc`, `def` produit la chaîne `abcdef` seule. Ceci peut être vu par l'évaluation de ce qui suit :

```
(concat 'abc' 'def')
```

La valeur produite en évaluant cette expression est «`abcdef`».

Une fonction comme sous-chaîne utilise à la fois une chaîne et des nombres comme arguments. La fonction renvoie une partie de la chaîne, une chaîne du premier argument. Cette fonction prend trois arguments. Son premier argument est la chaîne de caractères, les deuxième et troisième arguments sont des nombres qui indiquent le début et la fin de la sous-chaîne. Les nombres sont un comptage du nombre de caractères (ponctuation et espaces compris) depuis le début de la chaîne.

Par exemple, si vous évaluez la suivante :

```
(substring ''The quick brown fox jumped.'' 16 19)
```

vous verrez "`fox`" apparaître dans la zone d'écho. Les arguments sont la chaîne et les deux nombres.

Notez que la chaîne passée à `substring` est un atome, même si elle est composée de plusieurs mots séparés par des espaces. Lisp compte tout entre les deux guillemets dans le cadre de la chaîne, y compris les espaces. Vous pouvez penser la fonction `substring` comme une sorte «d'écraseur d'atome», car il faut un atome contraire à l'indivisible et en extraire une partie. Cependant, `substring` est seulement capable d'extraire une sous-chaîne à partir d'un argument qui est une chaîne, pas d'un autre type d'atome comme un nombre ou un symbole.

1.8.2 Un argument comme la valeur d'une variable ou d'une liste

Un argument peut être un symbole qui renvoie une valeur quand il est évalué. Par exemple, lorsque le symbole de remplissage par colonne lui-même est évalué, il renvoie un nombre. Ce nombre peut être utilisé pour une addition. Positionner le curseur après l'expression suivante et le type `C-x C-e` :

```
(+ 2 fill-column)
```

La valeur sera un nombre deux de plus que ce que vous obtenez en évaluant `fill-column` seul. Pour moi, c'est 74, parce que ma valeur de `fill-column` est 72.

Comme nous venons de le voir, un argument peut être un symbole qui renvoie une valeur lors de l'évaluation. En outre, un argument peut être une liste qui renvoie une valeur quand il est évalué. Par exemple, dans l'expression suivante, les arguments de la fonction `concat` sont les chaînes «The» et «red foxes.» et la liste `(number-to-string (+ 2 fill-column))`.

---

de dérivation, il est venu à signifier «d'affirmer d'une manière contre laquelle d'autres peuvent faire des contre affirmations», qui ont conduit à la signification du mot comme une dispute. (Notons ici que le mot anglais a deux définitions différentes qui s'y rattachent en même temps. En revanche, dans Emacs Lisp, un symbole ne peut pas avoir deux définitions de fonctions différentes en même temps.)

```
(concat ''The '' (number-to-string (+ 2 fill-column)) '' red foxes.'')
```

Si vous évaluez cette expression et si, comme avec mon Emacs, `fill-column` évalue à 72—«The 74 red foxes.» apparaît dans la zone écho. (Notez que vous devez mettre des espaces après le mot «The» et avant le mot «red» de sorte qu'ils apparaissent dans la chaîne finale. La fonction `number-to-string` convertit l'entier que la fonction d'addition renvoie à une chaîne. `number-to-string` est également connu comme `int-to-string`.)

1.8.3 Nombre variable d'arguments

Certaines fonctions, comme `concat`, `+` ou `*`, prendre n'importe quel nombre d'arguments. (Le `*` est le symbole de multiplication.) Ceci peut être vu par l'évaluation de chacune des expressions suivantes de la manière habituelle. Ce que vous verrez dans la zone d'écho est imprimé dans ce texte après «⇒», que vous pouvez lire comme «évalué à ».

Dans la première série, les fonctions n'ont pas d'arguments :

```
(+)  ⇒  0
(*)  ⇒  1
```

Dans cet ensemble, les fonctions ont un argument de chaque :

```
(+ 3)  ⇒  3
(* 3)  ⇒  3
```

Dans cet ensemble, les fonctions ont trois arguments de chaque :

```
(+ 3 4 5)  ⇒  12
(* 3 4 5)  ⇒  60
```

1.8.4 Utiliser le mauvais type d'objet pour un argument

Quand une fonction est passée un argument de type incorrect, l'interprète Lisp produit un message d'erreur. Par exemple, la fonction `+` attend les valeurs de ses arguments pour être des nombres. Comme une expérience, nous pouvons passer le symbole cité bonjour au lieu d'un nombre. Positionner le curseur après l'expression suivante et le type `C-x C-e` :

```
(+ 2 'hello)
```

Lorsque vous faites cela, vous allez générer un message d'erreur. Ce qui est arrivé est que `+` a essayé d'ajouter le 2 à la valeur renvoyée par `'hello`, mais la valeur renvoyée par `'hello` est le symbole `hello`, pas un nombre. Seuls des chiffres peuvent être ajoutés. Donc `+` ne pouvait pas mener à bien son addition.

Vous aller créer et entrer dans un tampon `*Backtrace*` qui dit :

```
Debugger entered--Lisp error :

               (wrong-type-argument number-or-marker-p hello)

+(2 hello)
   eval((+ 2 (quote hello)))
   eval-last-sexp-1(nil)
   eval-last-sexp(nil)
   call-interactively(eval-last-sexp)
```

Comme d'habitude, le message d'erreur essaie d'être utile et logique après vous apprenez à lire[6].

La première partie du message d'erreur est simple ; il est dit «`wrong type argument`». Vient ensuite le mot de jargon mystérieux «`number-or-marker-p`». Ce mot est d'essayer de vous dire ce genre d'argument du `+` prévu.

Le symbole `number-or-marker-p` dit que l'interprète Lisp essaie de déterminer si l'information a présenté (la valeur de l'argument) est un nombre ou un marqueur (un objet spécial représentant une position de tampon). Ce qu'il fait est le test pour voir si le `+` est étant des nombres à ajouter. Il teste aussi de voir si l'argument est quelque chose appelé un marqueur, qui est une caractéristique spécifique de Emacs Lisp. (Dans Emacs, les emplacements dans une mémoire tampon sont enregistrés en tant que marqueurs. Lorsque la marque est réglée avec le `C-@` ou la commande `C-SPC`, sa position est maintenue en tant que marqueur. La marque peut être considérée comme un nombre—le nombre de caractères les emplacement est à partir du début de la mémoire tampon). En Lisp, `+` peut être utilisé pour ajouter la valeur numérique de positions de marqueurs que des nombres.

Le «`p`» de «`number-or-marker-p`» est l'incarnation d'une pratique a commencé dans les premiers jours de la programmation Lisp. Le «`p`» signifie «prédicat». Dans le jargon, utilisé par les chercheurs Lisp, un prédicat renvoie à une fonction pour déterminer si une propriété est vraie ou fausse. Ainsi, le «`p`» nous dit que ce `number-or-marker-p` est le nom d'une fonction qui détermine si c'est vrai ou faux que l'argument fourni est un nombre ou un marqueur. Autres symboles Lisp qui se terminent par «`p`» comprennent `zerop`, une fonction qui teste si son argument a la valeur de zéro, et `listp`, une fonction qui teste si son argument est une liste.

Enfin, la dernière partie du message d'erreur est le symbole `hello`. C'est la valeur de l'argument qui a été adopté pour `+`. Si l'addition avait été adoptée le bon type d'objet, la valeur passée aurait été un certain nombre, comme 37, plutôt qu'un symbole comme `hello`. Mais alors vous n'auriez pas reçu le message d'erreur.

1.8.5 La fonction `message`

Comme `+`, la fonction de `message` prend un nombre variable d'arguments. Elle est utilisée pour envoyer des messages à l'utilisateur et est si utile que nous allons la décrire ici.

Un message est imprimé dans la zone écho. Par exemple, vous pouvez imprimer un message dans votre zone de répercussion en évaluant la liste suivante :

```
(message ``This message appears in the echo area'')
```

L'ensemble de la chaîne de caractères entre guillemets est un argument unique et est imprimé dans sa totalité. (Notez que dans cet exemple, le message lui-même apparaît dans la zone écho entre guillemets ; cela parce que vous voyez la valeur retournée par la fonction `message`. Dans la plupart des utilisations de `message` dans les programmes que vous écrivez, le texte sera imprimé dans la zone d'écho comme un effet secondaire, sans les guillemets. Voir Section 3.3.1 "multiply-by-seven in detail" page 29, pour un exemple de cela).

Cependant, s'il y a un '`%s`' dans la chaîne de caractères entre guillemets, la fonction `message` n'imprimera pas le '`%s`' en tant que tel, mais l'assimile à un argument qui suit la chaîne. Elle évalue le second argument et imprime sa valeur à l'endroit où se situe '`%s`' dans la chaîne.

Vous pouvez voir cela en positionnant le curseur après l'expression suivante et en tapant `C-x C-e` :

```
(message ``The name of this buffer is : %s.'' (buffer-name))
```

Dans Info, `` `The name of this buffer is :*info*.' `` apparaîtra dans la zone d'écho. La fonction `buffer-name` renvoie le nom du tampon comme une chaîne, où la fonction `message`

[6] `(quote hello)` est une expansion de l'abréviation `'hello`

insère à la place de `'%s'`.

Pour imprimer une valeur comme un entier, utiliser `'%d'` de la même manière que `'%s'`. Par exemple, pour imprimer un message dans la zone écho qui indique la valeur de `fill-column`, évaluer les points suivants :

```
(message ``The value of fill-column is : %d.'' fill-column)
```

Sur mon système, lorsque j'évalue cette list, ``` ``The value of fill-column is 72.'' ``` apparaît dans ma zone d'écho.[7]

S'il y a plus qu'un `'%s'` dans la chaîne entre les quotes, la valeur du premier argument est affichée à l'endroit du premier `'%s'`, la valeur du deuxième argument est affichée à l'endroit du deuxième `'%s'` et ansi de suite.

Par exemple, si vous faites l'évaluation suivante :

```
(message ``There are %d %s in the office!''
         (- fill-column 14) ``pink elephants'')
```

un message plutôt lunatique apparaîtra dans votre zone de répercussion. Sur mon système, il dit,

```
``The are 58 pink elephants in the office!''.
```

L'expression `(- fill-column 14)` est évaluée et le nombre résultant est inséré à la place de `'%d'` ; et la chaîne de caractères ``` ``pink elephants'' ```, est traitée comme un argument simple et insérée à la place de `'%s'` :

```
(message  ''He saw %d %s''
          (- fill-column 32)
          (concat ``red ``
                              (substring
                              ``The quick brown foxes jumped.'' 16 21)
                              `` leaping.'')))
```

Dans cet exemple, la fonction `message` a trois arguments : la chaîne ``` ``He saw %d %s'' ```, l'expression `(- fill-column 32)` est insérée à la place de `'%d'` ; et la valeur renvoyée par l'expression commençant par `concat` est insérée à la place de `'%s'`.

Lorsque votre colonne de remplissage est de 70 et que vous évaluer l'expression, le message ``` ``He saw 38 red foxes leaping.'' ``` aparaît dans votre zone d'écho.

1.9 Initialisation de variable

Il y a plusieurs façons d'affecter une variable. Une des façons est d'utiliser la fonction `set` ou `setq`. Une autre façon est d'utiliser la fonction `let` (voir Section 3.6 “let” page 31). (Le jargon pour ce processus est de *lier* une variable à une valeur.)

Les sections suivantes ne décrivent pas seulement le fonctionnement de `set` et `setq` mais illustrent aussi comment les arguments sont passés.

1.9.1 Utilisation de `set`

Pour affecter la valeur du symbole `flowers` à la liste `'(rose violet daisy buttercup)`, évaluer l'expression suivante en positionnnant le curseur après l'expression et en tappant le raccourci suivant : `C-x C-e`.

```
(set 'flowers '(rose violet daisy buttercup))
```

[7] En fait vous pouvez utiliser `'%s'` pour afficher un nombre. Ce n'est pas spécifique. `%d` n'affiche que la partie à gauche de la virgule d'un nombre décimal, et rien d'autre qui ne soit un nombre.

Cette liste `(rose violet daisy buttercup)` apparaîtra dans la zone d'écho. C'est ce qui est *renvoyé* par la fonction `set`. Comme un effet secondaire, le symbole `flowers` est lié à la liste ; qui est, le symbole `flowers` qui peut être considéré comme une variable, est donné à la liste comme sa valeur. (Ce processus, par la manière, illustre comment un effet secondaire à l'interprète Lisp, réglage de la valeur, peut être l'effet principal qui nous intéresse nous les humains. C'est parce que chaque fonction Lisp doit renvoyer une valeur si elle ne reçoit pas une erreur, mais elle aura seulement un effet secondaire si elle est conçue pour avoir un.)

Après l'évaluation de l'expression `set`, vous pouvez évaluer le symbole `flowers` et cela vous renverra la valeur que vous venez de régler. Voici le symbole. Placez le curseur après lui et tapez la séquence de touche `C-x C-e`.

```
flowers
```

Quand vous évaluez `flowers`, la liste `(rose violet daisy buttercup)` apparaît dans la zone d'écho.

Incidemment, si vous évaluez `'flowers`, la variable avec une quote devant elle, ce que vous verrez dans la zone d'écho c'est le symbole lui-même. Voici le symbole quoté, donc vous pouvez essayer ça :

```
'flowers
```

Notez aussi, que lorsque vous utilisez aussi `set`, vous avez besoin de quotes pour chaque argument de `set`, à moins que vous ne souhaitiez qu'ils soient évalués. Puisque nous ne voulons pas non plus évalué les arguments ni de la variable `flowers` ni de la liste `(rose violet daisy buttercup)`, les deux sont quotés. (Lorsque vous utilisez `set` sans quote pour son premier argument, le premier argument est evalué avant que n'importe quoi d'autre soit fait. Si vous aviez fait ça et que `flowers` n'avait pas encore de valeur, vous auriez reçu un message d'erreur disant que `'Symbol's value as variable is void'` ; d'autre part, si `flowers` avait renvoyé une valeur après qu'elle ait été évalué, le `set` tenterait de régler la valeur qui a été renvoyé. Il y a des situations où c'est la bonne chose à faire pour la fonction ; mais de telles situations sont rares.)

1.9.2 Utilisation de `setq`

En pratique, vous mettez une quote presque toujours devant le premier argument de `set`. La combinaison de `set` et d'une quote pour le premier argument et si fréquente qu'elle a son propre nom : la forme spéciale `setq`. Cette forme spéciale est comme `set` sauf que le premier argument a automatiquement une quote, donc vous n'avez pas besoin de tapez une quote vous-même. En outre, par commodité, `setq` vous permez de régler plusieurs variables différentes avec différentes valeurs, toutes en une expression.

Pour affecter la variable `carnivores` de la liste `'(lion tiger leopard)` en utilisant `setq`, l'expression suivante est utilisée :

```
(setq carnivores '(lion tiger leopard))
```

Avec `set`, l'expression ressemblerait à ça :

```
(set 'carnivores '(lion tiger leopard))
```

En outre, `setq` peut être utlisée pour assigner différentes valeurs à différentes variables. Le premier argument est lié à la valeur du second argument, le troisième est lié à la valeur du quatrième, et ainsi de suite. Par exemple, vous pouvez utiliser ce qui suit pour affecter une liste d'arbres au symbole `trees` et une liste d'herbivores au symbole `herbivores` :

```
(setq trees '(pine fir oak maple)
herbivores '(gazelle antelope zebra))
```

(L'expression aurait aussi bien pu être sur une seule ligne, mais ça n'aurait pas été bien mis en page ; et les humains trouvent qu'il est plus simple de lire les listes qui sont bien mises en page.)

Même si j'utilise le terme 'assigner', il y a une autre façon de penser le travail de `set` et `setq` ; et c'est de dire que `set` et `setq` mettent le symbole *point* dans la liste. Cette dernière façon de penser est très communes et dans les chapitres à venir nous verrons au moins un symbole qui a 'pointer' comme partie de son nom. Le nome est choisi parcee le symbole a une valeur, spécifiquement une liste, attachée à lui ; ou, dit autrement, le symbole est réglé pour «pointer» sur la liste.

1.9.3 Compteurs

Voici un exemple qui montre comment utiliser `setq` dans un compteur. Vous devez utiliser cela pour compter combien de fois une partie de votre programme se répète. D'abord initialiser une variable à zéro ; ensuite ajouter un au nombre chaque fois que le programme se répète. Pour faire cela, vous avez besoin d'une variable qui sert à compter, et de deux expressions : une expression `setq` initiale qui affecte le compteur à zéro ; et une seconde expression `setq` qui incrémente le compteur chaque fois qu'elle est évaluée.

```
(setq counter 0)                ; appelons-le l'initialisateur
(setq counter (+ counter 1))    ; c'est l'incrémenteur
counter                         ; c'est le compteur
```

(Le texte suivant le ';' sont des commentaires. Voir Section 3.2.1 "Change a Function Definition", page 28.)

Si vous évaluez la première de ces expressions, l'initialisateur, `(setq counter 0)`, et ensuite évaluez la troisième expression, `counter`, le nombre 0 apparaît dans la zone d'écho. Si vous évaluez ensuite la seconde expression, l'incrémenteur, `(setq counter (+ counter 1))`, le compteur aura la valeur 1. Donc si vous évaluez `counter`, le nombre 1 apparaîtra dans la zone d'écho. Chaque fois que évaluez la seconde expression, la valeur du compteur sera incrémentée.

Quand vous évaluez l'incrémenteur, `(setq counter (+ counter 1))`, l'interprète Lisp évalue d'abord la liste la plus interne ; c'est l'addition. Afin d'évaluer cette liste, il doit évaluer la variable `counter` et le nombre 1. Quand il évalue la variable `counter`, il reçoit sa valeur courante. Il passe cette valeur et le nombre 1 à la fonction `+` qui les ajoute ensemble. La somme est ensuite renvoyée à la valeur de la liste qui la contient et passée à la fonction `setq` qui affecte la variable `counter` de cette nouvelle valeur. Ainsi la valeur de la variable, `counter`, est modifiée.

1.10 Résumé

L'apprentissage de Lisp est comme l'escalade d'une colline dans laquelle la première partie est la plus dure. Vous avez maintenant grimpé la partie la plus difficile ; ce qui reste deviendra plus facile maintenant que vous avez progressé.

En résumé,

- Les programmes Lisp sont constitués d'expressions, qui sont des listes ou des atomes individuels.
- Les listes sont constituées de 0 ou plusieurs atomes ou de listes internes, séparées par des espaces et entourées par des parenthèses. Une liste peut être vide.
- Les atomes sont des symboles multi-caractères, comme `forward-paragraph`, symboles caractère simple comme `+`, chaînes de caractères entre guillemets, ou des nombres.
- Un nombre s'évalue lui-même.

- Une chaîne entre guillemets s'évalue également elle-même.
- Lorsque vous évaluez un symbole par lui-même, sa valeur est renvoyée.
- Lorsque vous évaluez une liste, l'interprète Lisp regarde d'abord le premier symbole dans la liste et ensuite la fonction liée à ce symbole. Ensuite les instructions dans la définition de fonction sont effectuées.
- Un guillemet simple (quote), `'`, dit à l'interprète Lisp qu'il devrait renvoyer l'expression suivante comme écrite, et ne pas l'évaluer comme si la quote n'était pas là.
- Les arguments sont des information passées à la fonction. Les arguments d'une fonction sont calculés par évaluation du reste des éléments de la liste dont la fonction est le premier élément.
- Une fonction renvoie toujours une valeur quand elle est évaluée (à moins d'obtenir une erreur) ; en outre, il peut aussi effectuer une action appelée «effet secondaire». Dans de nombreux cas, le but principal d'une fonction est de créer un effet secondaire.

1.11 Exercices

Quelques exercices simples :

- Générer un message d'erreur en évaluant un symbole approprié qui n'est pas entre parenthèses. (Par exemple évaluer `+`)
- Générer un message d'erreur en évaluant un symbole approprié entre parenthèses. (Par exemple évaluer `(+ 2 bonjour)`)
- Créer un compteur qui incrémente par deux au lieu de un. (Par exemple

  ```
  (setq counter 0)
  setq counter (+ counter 2)
  counter
  ```

- Écrire une expression qui affiche un message dans la zone d'écho quand il est évalué. (Par exemple `(message ``Ceci est un message dans le tampon : %s.'' (buffer-name))`

Chapitre 2

Évaluation pratique

Avant d'apprendre à écrire une définition de fonction dans Emacs Lisp, il est utile de passer un peu de temps à évaluer diverses expressions qui ont déjà été écrites. Ces expressions seront des listes avec des fonctions pour premier (et souvent seul) élément. Puisque certaines des fonctions associées avec des tampons sont à la fois simples et intéressantes, nous allons commencer avec elles. Dans cette section, nous allons en évaluer quelques-unes. Dans une autre section, nous étudierons le code de plusieurs autres fonctions reliées à des tampons, pour voir comment elles ont été écrites.

Chaque fois que vous donnez une commande d'édition à Emacs Lisp, comme la commande pour déplacer le curseur ou pour faire défiler l'écran, *vous évaluez une expression*, le premier élément est une fonction. *C'est comme cela qu'Emacs fonctionne.*

Lorsque vous tapez des touches, vous provoquez l'interprète Lisp pour évaluer une expression et c'est comme cela que vous obtenez un résultat. Même taper le texte brut implique une évaluation d'une fonction Emacs Lisp, dans ce cas, celle qui utilise `self-insert-command`, qui insère simplement le caractère que vous avez saisi. Les fonctions que vous évaluez en tapant des raccourcis sont appelées fonctions *interactives*, ou *commandes* ; comment vous faire une fonction interactive sera illustré dans le chapitre comment écrire des définitions de fonctions. Voir Section 3.3 "Faire une fonction interactive", page 28.

En plus de la saisie des commandes clavier, nous avons vu une seconde façon d'évaluer une expression : en positionnant le curseur après une liste et en tapant `C-x C-e`. C'est ce que nous allons faire dans le reste de cette section. Il y a d'autres façons d'évaluer une expression ; celles-ci seront décrites lorsque nous viendront à elles.

En plus d'être utilisées pour pratiquer l'évaluation, les fonctions présentées dans les prochaines sections sont importantes en elles-mêmes. Une étude de ces fonctions opère une nette distinction entre les tampons et les fichiers, comme changer de tampon, et comment déterminer un emplacement à l'intérieur.

2.1 Noms des tampons

Les deux fonctions, `buffer-name` et `buffer-file-name`, montrent la différence entre un fichier et un tampon. Lorsque vous évaluez l'expression suivante, `(buffer-name)`, le nom du tampon apparaît dans la zone d'écho. Quand vous évaluez `(buffer-file-name)`, le nom du fichier auquel se réfère le tampon est affiché dans la zone d'écho. Habituellement, le nom renvoyé par `(buffer-name)` est le même que le nom de fichier auquel il fait référence, et le nom renvoyé par `(buffer-file-name)` est le nom de chemin complet du fichier.

Un fichier et un tampon sont deux entités différentes. Un fichier est une information enregistrée de façon permanente dans l'ordinateur (à moins que vous ne l'effaciez). Un tampon, d'un autre côté, est une information à l'intérieur d'Emacs qui va disparaître à la fin de la session (ou quand vous «tuez» le tampon). Habituellement, un tampon contient des informations que vous avez copié à partir d'un fichier ; nous disons que le tampon *visite* ce fichier. Cette copie est ce que vous travaillez et modifiez. Les changements du tampon ne change pas le fichier, jusqu'à ce que vous enregistrez le tampon. Lorsque vous sauvegardez le tampon, le tampon est copié dans le fichier et est donc sauvegardé de façon permanente.

Si vous lisez ceici dans Info à l'intérieur de GNU Emacs, vous pouvez évaluer chacune des expressions suivantes en positionnant le curseur après et en tapant `C-x C-e`.

```
(buffer-name)
(buffer-file-name)
```

Quand je fais cela dans Info, la valeur renvoyée par l'évaluation de `(buffer-name)` est `''*info*''`, et la valeur renvoyée par `(buffer-file-name)` est `nil`.

D'autre part, pendant que je vous écris ce document, la valeur renvoyée par l'évaluation de `(buffer-name)` est `''introduction.texinfo''`, et la valeur renvoyée par l'évaluation de `(buffer-file-name)` est `''/gnu/work/intro/introduction.texinfo''`.

Le premier est le nom du tampon et le dernier est le nom du fichier. Dans Info, le nom du tampon est `''*info*''`. Info ne pointe vers aucun fichier, donc le résultat de l'évaluation de `(buffer-file-name)` est `nil`. Le symbole `nil` vient du mot Latin pour 'rien' ; dans ce cas, cela signifie que le tampon n'est associé à aucun fichier. (En Lisp, `nil` est aussi utilisé pour signifié 'faux' et est un synonyme pour la liste vide, `()`.)

Lorsque j'écris, le nom de mon tampon est `''introduction.texinfo''`. Le nom du fichier vers lequel il pointe est `''/gnu/work/intro/introduction.texinfo''`.

(Dans ces expressions, les parenthèses disent à l'interprète Lisp de traiter `buffer-name` et `buffer-file-name` comme des fonctions ; sans les parenthèses, l'interprète serait tenté d'évaluer les symboles comme des variables. Voir Section 1.7 "Variables", page 8.)

En dépit de la distinction entre les fichiers et les tampons, vous trouverez souvent que les gens se réfèrent à un fichier quand ils veulent dire tampon et vice-versa. En effet, la plupart des gens disent : «Je suis dans l'édition d'un fichier», plutôt que de dire : «Je suis dans l'édition d'un tampon que je vais bientôt enregistrer dans un fichier.» Il est presque toujours clair à partir du contexte de deviner ce que les gens veulent dire. Lorsque vous traitez avec des programmes informatiques, cependant, il est important de garder à l'esprit la distinction, puisque l'ordinateur n'est pas aussi intelligent qu'une personne (pour deviner).

Le mot «tampon», en passant, vient de la signification du mot coussin qui amortit la force d'une collision. Dans les premiers ordinateurs, un tampon amortissait l'interaction entre les fichiers et l'unité centrale de traitement de l'ordinateur. Les tambours ou bandes qui tenaient un fichier et l'unité centrale de traitement étaient des pièces d'équipement qui étaient très différentes les unes des autres, à travailler à leurs propres vitesses, par à-coups. Le tampon a rendu possible pour eux de travailler ensemble efficacement. Finalement, le tampon est passé du statut d'intermédiaire, un lieu de détention temporaire, à celui de lieu où le travail se fait. Cette transformation est un peu comme celle d'un petit port qui a grandi dans une grande ville : au début il était simplement le lieu où des marchandises étaint entreposées temporairement avant d'être chargées sur les bateaux ; puis il est devenu un centre commercial et culturel.

Pas tous les tampons sont associés à des fichiers. Par exemple, un tampon `*scratch*` ne visite aucun fichier. De même, un tampon `*Help*` n'est associé à aucun fichier.

Dans les anciens temps, lorsque vous aviez manqué un fichier `/.emacs` et commencé une session Emacs en tapant la commande `emacs` seule, sans nommer aucun fichier, Emacs démarrait avec le tampon `*scratch*`. De nos jours, vous verrez un écran de démarrage. Vous pouvez suivre

l'une des commandes proposées sur l'écran de démarrage, visiter un fichier ou appuyer sur la barre d'espace pour atteindre le tampon `*scratch*`.

Si vous passez au tampon `*scratch*`, tapez `(buffer-name)`, positionnez le curseur à la suite, et tapez `C-x C-e` pour évaluer l'expression. Le nom `''*scratch*''` sera renvoyé et apparaîtra dans la zone d'écho. `''*scratch*''` est le nom du tampon. Quand vous tapez `(buffer-file-name)` dans le tampon `*scratch*` et évaluez ça, `nil` apparaît dans la zone d'écho, juste comme ça le fait lorsque vous évaluez `(buffer-file-name)` dans Info.

Incidemment, si vous êtes dans le tampon `*scratch*` et souhaitez que la valeur renvoyée par une expression apparaisse dans le tampon `*scratch*` lui-même au lieu de la zone d'écho, tapez `C-u C-x C-e` au lieu de `C-x C-e`. Cela provoque l'apparition de la valeur renvoyée après l'expression. Le tampon ressemblera à ça :

```
(buffer-name)''*scratch*''
```

Vous ne pouvez pas faire ça dans Info puisque Info est en lecture seule et ne vous autorisera pas à changer le contenu de son tampon. Mais vous pouvez faire cela dans n'importe quel tampon que vous éditez ; et quand vous écrivez du code ou une documentation (comme ce livre), c'est fonction sont très utiles.

2.2 Obtenir un tampon

La fonction `buffer-name` renvoie le *nom* du tampon ; pour obtenir le tampon *lui-même*, une fonction différente est nécessaire : la fonction `current-buffer`. Si vous utilisez cette fonction dans un code, ce que vous obtenez est le tampon lui-même.

Un nom d'objet et l'objet ou l'entité à laquelle se réfère le nom sont différents entre eux. Vous n'êtes pas votre nom. Vous êtes une personne à laquelle les autres font référence par votre nom. Si vous demandez à parler à George et quelqu'un vous donne une carte avec les lettres `'G', 'e', 'o', 'r', 'g', 'e'` écrites dessus, vous serez peut être amusé, mais vous ne serez pas satisfait. Vous ne voulez pas parler au nom, mais à la personne à laquelle le nom fait référence. Un tampon est similaire : le nom du tampon scratch est `*scratch*`, mais le nom n'est pas le tampon. Pour obtenir le tampon lui-même vous avez besoin d'utiliser une fonction comme `current-buffer`.

Cependant, il y a une légère complication : si vous évaluez `current-buffer` dans une expression, comme nous le ferons ici, ce que vous voyez est une représentation imprimée du nom du tampon sans le contenu du tampon. Emacs travaille de cette façon pour deux raisons : la tampon peut être des milliers de lignes de long—trop long pour être affiché ; et, un autre tampon peut avoir le même contenu mais un nom différent, et il est important de pouvoir les distinguer.

Voici une expression contenant la fonction :

```
(current-buffer)
```

Si vous évaluez cette expression dans Info dans Emacs de la façon habituelle, `#<buffer *info*>` apparaîtra dans la zone d'écho. Le format spécial indique que le tampon lui-même est renvoyé, plutôt que juste son nom.

Incidemment, pendant que vous pouvez taper un nombre ou symbole dans un programme, vous ne pouvez pas faire ça avec une représentation imprimée d'un tampon : la seule façon d'obtenir un tampon lui-même c'est avec une fonction telle que `current-buffer`.

Une fonction liée est `other-buffer`. Elle renvoie le tampon le plus récemment sélectionné autre que celui sur lequel vous êtes en ce moment, pas une représentation imprimée de son nom. Si vous avez récemment changé depuis le tampon `*scratch*`, `other-buffer` renverra ce tampon.

Vous pouvez voir ça en évaluant l'expression :

```
(other-buffer)
```

Vous devriez voir `#<buffer *scratch*>` apparaître dans la zone d'écho, ou le nom de n'importe quel autre tampon que vous avez changé récemment.[1]

2.3 Changer de tampon

La fonction `other-buffer` fournit en fait un tampon quand il est utilisé comme un argument pour une fonction qui en exige un. Nous pouvons voir cela en utilisant `other-buffer` et `switch-to-buffer` pour passer à un tampon différent.

Mais d'abord, une brève introduction à la fonction `switch-to-buffer`. Lorsque vous passez du tampon Info à celui de `*scratch*` pour évaluer `(buffer-name)`, vous avez probablement tapé `C-x b` puis `*scratch*`[2] lorsque vous êtiez invité à saisir le nom du tampon vers lequel vous souhaitiez aller dans le mini-tampon. Le raccourci, `C-x b`, provoque l'interprète Lisp pour évaluer la fonction interactive `switch-to-buffer`. Comme on l'a dit plus tôt, voilà comment Emacs fonctionne : différent raccourcis appellent ou exécutent différentes fonctions. Par exemple, `C-f` appelle `forward-char`, `M-e` appelle `forward-sentence`, et ainsi de suite.

En écrivant `switch-to-buffer`, et en lui donnant un tampon pour basculer, nous pouvons changer de tampon juste de la même façon que fait `C-x b` :

```
(switch-to-buffer (other-buffer))
```

Le symbole `switch-to-buffer` est le premier élément de la liste, de sorte que l'interprète Lisp va le traiter une fonction et exécuter les instructions qui y sont attachées. Mais avant de faire cela, l'interprète notera que `other-buffer` est entre parenthèses à l'intérieur et le travail sur ce premier symbole. `other-buffer` est le premier (et dans ce cas, le seul) élément de cette liste, donc l'interprète Lisp appelle et exécute la fonction. Cela renvoie un autre tampon. Ensuite, l'interprète lance `switch-to-buffer`, en lui passant, comme un argument, l'autre tampon, qui est celui que Emacs basculera vers. Si vous lisez ceci dans Info, essayez maintenant. Évaluer l'expression. (Pour revenir, tapez `C-x b RET`.)[3]

Dans les exemples de programmation dans les sections suivantes de ce document, vous verrez la fonction `set-buffer` plus souvent que `switch-to-buffer`. Ceci est dû à une différence entre les programmes et les humains : les humains ont des yeux et s'attendent à voir le tampon sur lequel ils travaillent sur leurs terminaux. Cela est si évident, que cela va presque sans dire. Toutefois, les programmes n'ont pas d'yeux. Quand un programme fonctionne sur un tampon, ce tampon n'a pas besoin d'être visible sur l'écran.

`switch-to-buffer` est conçu pour l'homme et fait deux choses différentes : il change vers le tampon sur lequel l'attention de Emacs est dirigé ; et il commute le tampon affiché dans la fenêtre pour le nouveau tampon. `set-buffer`, d'autre part, ne fait qu'une chose : il passe l'attention du programme à un tampon différent. Le tampon de l'écran reste inchangé (bien sûr, rien ne se passe normalement jusqu'à ce que les commandes finissent leurs exécutions).

---

[1] En fait, par défaut, si le tampon à partir duquel vous venez de changer est visible pour vous dans une autre fenêtre, un autre tampon sera choisi, le plus récent que vous ne pouvez pas voir ; ceci est une subtilité que j'oublie souvent.

[2] Ou plutôt, économiser la frappe, vous avez probablement tapé `RET` si le tampon par défaut était `*scratch*`, ou si c'était différent, alors vous avez juste tapé une partie du nom, telle que `*sc`, pressé la touche `TAB` pour provoquer la complétion, et ensuite tapé `RET`

[3] Rappelez-vous, cette expression vous déplacera dans l'autre tampon le plus récent que vous ne pouvez pas voir. Si vous voulez vraiment aller vers le tampon le plus récent sélectionné, même si vous pouvez le voir, vous avez besoin d'évaluer l'expression plus complexe suivante :

```
(switch-to-buffer (other-buffer (current-buffer) t))
```

Dans ce cas, le premier argument de `other-buffer` lui dit quel tampon à sauter—celui en court—et le second argument inqique à `other-buffer` que c'est OK de changer vers un tampon visible. En utilisation régulière, `switch-to-buffer` vous amène une fenêtre invisible puisque vous utilisez le plus souvent `C-x o (other-window)` pour aller à un autre tampon visible.

Aussi, nous venons d'introduire un autre terme de jargon, le mot *appel*. Lorsque vous évaluez une liste dans laquelle le premier symbole est une fonction, vous appellez cette fonction. L'utilisation du terme provient de la notion de la fonction comme une entité qui peut faire quelque chose pour vous si vous 'l'appelez'—juste comme un plombier est une entité qui peut réparer une fuite si vous l'appelez.

2.4 Taille du tampon et localisation du point

Enfin, regardons plusieurs fonctions assez simples, `buffer-size, point-min` et `point-max`. Celles-ci donnent des informations sur la taille d'un tampon et l'emplacement du point en son sein.

La fonction `buffer-size` vous indique la taille du tampon courant ; autrement dit, la fonction renvoie un compte du nombre de caractères dans le tampon.

```
(buffer-size)
```

Vous pouvez évaluer cela de façon habituelle, en positionnant le curseur après l'expression et en tapant `C-x C-e`.

Dans Emacs, la position actuelle du curseur est appelée le *point*. L'expression `(point)` renvoie un nombre qui vous indique où se trouve le curseur comme un comptage de nombre de caractères à partir du début du tampon jusqu'au point.

Vous pouvez voir le nombre de caractères pour le point dans ce tampon en évaluant l'expression suivante de la manière usuelle :

```
(point)
```

Pendant que j'écris ceci, la valeur du point est 65724. La fonction `point` est utilisée fréquemment dans certains exemples plus loin dans ce livre.

La valeur du point dépend, bien sûr, de son emplacement dans le tampon. Si vous évaluez le point à cet endroit, le nombre sera plus grand :

```
(point)
```

Pour moi, la valeur du point à cet endroit est 66043, ce qui signifie qu'il y a 319 caractères (espaces compris) entre les deux expressions. (Sans doute, vous verrez des nombres différents, puisque j'aurais édité depuis ma première évaluation du point.)

La fonction `point-min` est quelque peu semblable à un `point`, mais elle renvoie la valeur de la valeur minimale admissible du point dans le tampon courant. C'est le nombre 1 sauf si le *réduction* est en vigueur. (La réduction est un mécanisme par lequel vous pouvez vous limiter, ou un programme, d'opérations sur seulement une partie d'un tampon. Voir le chapitre 6 "Réduction et élargissement", page 37.) De même, la fonction `point-max` renvoie la valeur de la valeur maximale admissible du point dans le tampon courant.

2.5 Exercice

Trouver un fichier avec lequel vous travaillez et progresser vers son milieu. Trouver son nom de tampon (`(buffer-name)`), nom de fichier (`(buffer-file-name)`), la longueur (`(buffer-size)`), et votre position (`(point)`) dans le fichier.

Chapitre 3

Comment écrire des définitions de fonctions

Lorsque l'interprète Lisp évalue une liste, il regarde si le premier symbole sur la liste a une définition de fonction attachée à lui ; ou, en d'autres termes, si le symbole pointe vers une définition de fonction. Dans le cas contraire, l'ordinateur exécute les instructions dans la définition. Un symbole qui a une définition de fonction est appelé, tout simplement, une fonction (même si, à proprement parler, la définition est la fonction et le symbole se réfère à elle.)

Toutes les fonctions sont définies en termes d'autres fonctions, à l'exception de quelques fonctions primitives qui sont écrites dans le langage de programmation C. Lorsque vous écrivez les définitions de fonctions, vous les écrirez en Emacs Lisp et utiliserez d'autres fonctions comme blocs de construction. Certaines des fonctions que vous utiliserez seront elles-mêmes écrites en Emacs Lisp (peut être par vous) et certaines seront des primitives écrites en C. Les fonctions primitives sont utilisées exactement comme celles écrites en Emacs Lisp et se comportent comme elles. Elles sont écrites en C afin que nous puissions facilement lancer GNU Emacs sur tout ordinateur d'une puissance suffisante pour faire fonctionner C.

Permettez-moi de souligner de nouveau ceci : quand vous écrivez du code en Emacs Lisp, vous ne distinguez pas celles écrites en C de celles écrites en Emacs Lisp. La différence est sans importance. Je mentionne la distinction seulement parce qu'il est intéressant de le savoir. En effet, à moins que vous ne l'étudiez, vous ne saurez pas si une fonction déjà écrite est écrite en Emacs Lisp ou en C.

3.1 La forme spéciale `defun`

En Lisp, un symbole tel que `mark-whole-buffer` a un code qui lui est attaché et qui dit à l'ordinateur que faire lorsque la fonction est appélée. Ce code est appelé la *définition de fonction* et est créé par l'évaluation d'une expression Lisp qui commence par le symbole `defun` (qui est une abréviation pour *define function*[1]). Parce que `defun` n'évalue pas ses arguments de la manière habituelle, elle est appelée une *forme spéciale*.

Dans les sections suivantes, nous allons examiner les définitions de fonctions à partir du code source Emacs, comme `mark-whole-buffer`. Dans cette section, nous allons décrire une définition de fonction simple de sorte que vous puissiez voir à quoi ça ressemble. Cette définition de fonction utilise l'arithmétique, car cela en fait un exemple simple. Certaines personnes détestent

[1] définition de fonction

les exemples utilisant l'arithmétique ; toutefois, si vous êtes une telle personne, ne désespérez pas. Pratiquement tout le code, que nous allons étudier par la suite de cette introdcution implique l'arithmétique ou les mathématiques. Les exemples concernent essentiellement du texte d'une manière ou d'une autre.

Une définition de fonction a jusqu'à cinq parties suivant le mot `defun` :

1. Le nom du symbole à laquelle la définition de fonction doit être jointe.

2. Une liste des arguments qui seront passés à la fonction. Si aucun argument ne seront passés à la fonction, ceci est une liste vide, `()`.

3. Une documentation décrivant la fonction. (Techniquement facultatif, mais fortement recommandé.)

4. Éventuellement, une expression pour rendre la fonction interactive de sorte que vous pouvez l'utiliser en tapant `M-x`, puis le nom de la fonction ; ou en tapant sur une touche ou une combinaison de touches appropriées.

5. Le code qui indique à l'ordinateur ce qu'il doit faire : le *corps* de la fonction.

Il est utile de penser aux cinq parties d'une définition de fonction comme étant organisé dans un modèle, avec des fentes pour chaque partie :

```
(defun function-name (arguments …)
``optional-documentation…''
(interactive argument-passing-info) ; optional
body…)
```

À titre d'exemple, voici le code pour une fonction qui multiplie son argument par 7. (Cet exemple n'est pas interactif. Voir la section 3.3 “Faire une fonction interactive”, page 28, pour cette information.)

```
(defun muliply-by-seven (number)
``Multiply NUMBER by seven.''
(* 7 number))
```

Cette définition commence par une parenthèse et le symbole `defun`, suivi du nom de la fonction.

Le nom de la fonction est suivie par une liste qui contient les arguments qui seront passés à la fonction. Cette liste est appelée la *liste des arguments*. Dans cet exemple, la liste ne comporte qu'un seul élément, le symbole, `number`. Lorsque la fonction est utilisée, le symbole sera lié à la valeur qui est utilisée comme argument de la fonction.

Au lieu de choisir le nombre de mot pour le nom de l'argument, j'aurais pu choisir un autre nom. Par exemple, je pourrais avoir choisi le mot `multicplicand`. J'ai pris le mot «nombre» parce qu'il dit bien quel type de valeur est destinée à cet emplacement ; mais je pourrais tout aussi bien avoir choisi le mot «multiplicande» pour indiquer le rôle que la valeur placée à cet endroit va jouer dans le fonctionnement de la fonction. J'aurais pu l'appeler `foogle`, mais cela aurait été un mauvais choix, car il ne renseignerait pas sur sa signification. Le choix du nom revient au programmeur et doit être fait pour rendre le sens de la fonction le plus clair possible.

En effet, vous pouvez choisir le nom que vous souhaitez pour un symbole dans une liste d'arguments, même le nom d'un symbole utilisé dans une autre fonction : le nom que vous utilisez dans une liste d'arguments est réservé à cette définition particulière. Dans cette définition, le nom se réfère à une entité différente de n'importe qu'elle utilisation du même nom en dehors de cette définition de fonction. Supposons que vous avez un surnom «shorty» dans votre famille ; lorsque un membre de votre famille se réfère à «shorty», il parle de vous. Mais en dehors de votre famille,

dans un film, par exemple, le nom «shorty» se réfère à quelqu'un d'autre. Parce qu'un nom dans une liste d'argument est réservé à cette définition de fonction, vous pouvez changer la valeur d'un tel symbole à l'intérieur du corps d'une fonction sans changer sa valeur en dehors de la fonction. L'effet est similaire à celui produit par une expression `let`. (Voir Section 3.6 "let", page 31.)

La liste des arguments est suivie par la documentation qui décrit la fonction. Ceci est ce que vous voyez lorsque vous tapez `C-h f` et le nom d'une fonction. Incidemment, lorsque vous écrivez une documentation de ce genre, vous devriez faire la première ligne une phrase complète depuis quelques commandes, comme `apropos`, imprimer uniquement la première ligne d'une documentation multi-ligne. En outre, vous ne devriez pas indenter la deuxième ligne d'une documentation, si vous en avez une, parce que ça aurait l'air bizarre quand vous utilisez `C-h f (describe-function)`. La documentation est facultative, mais elle est si utile, elle devrait être inclus dans presque toutes les fonctions que vous écrivez.

La troisième ligne de l'exemple est constituée du corps de la définition de fonction. (La plupart des définitions de fonctions, bien sûr, sont plus longues que celle-là.) Dans cette fonction, le corps est la liste, `(* 7 number)`, qui dit de multiplier la valeur du nombre de 7. (Dans Emacs Lisp, `*` est la fonction de multiplication, comme `+` est la fonction d'addition.)

Lorsque vous utilisez la fonction `multiply-by-seven`, l'argument `number` évalue le nombre que vous avez utilisé. Voici un exemple qui montre comment `multiply-by-seven` est utilisé ; mais n'essayez pas encore de l'évaluer !

```
(multiply-by-seven 3)
```

Le symbole `number`, spécifié dans la définition de fonction de la prochaine section, est donné ou «lié à» la valeur 3 dans cet exemple d'utilisation. Notez que bien que `number` était entre parenthèses à l'intérieur dans la définition de fonction, l'argument passé à la fonction `(multiply-by-seven 3)` n'est pas entre parenthèses. Les parenthèses sont écrites dans la définition de fonction afin que l'ordinateur comprenne où se termine la liste d'arguments et le reste de la définition commence.

Si vous évaluez cet exemple, vous êtes susceptible d'obtenir un message d'erreur. (Allez-y, essayez !) C'est parce que nous avons écrit la définition de fonction, mais pas encore dit à l'ordinateur à propos de la définition—nous n'avons pas encore installé (ou «chargé») la définition de fonction dans Emacs. L'installation d'une fonction est le processus qui dit à l'interprète Lisp qu'elle est la définition de fonction. L'installation est décrite dans la section suivante.

3.2 Installer une définition de fonction

Si vous lisez ceci à l'intérieur d'Info dans Emacs, vous pouvez essayer la fonction `multiply-by-seven` en évaluant d'abord la définition de fonction puis en évaluant `(multiply-by-seven 3)`. Une copie de la définition de la fonction suivra. Placez le curseur après la dernière parenthèse de la définition de fonction et tapez `C-x C-e`. Lorsque vous faites cela, `multiply-by-seven` apparaîtra dans la zone d'écho. (Ce que cela signifie c'est que lorsqu'une définition de fonction est évaluée, la valeur renvoyée est le nom de la fonction définie.) En même temps, cette action installe la définition de fonction.

```
(defun multiply-by-seven (number)
 ``Multiply NUMBER by seven.''
 (* 7 number))
```

En évaluant ce `defun`, vous venez d'installer `multiply-by-seven` dans Emacs. La fonction fait désormais autant partie de Emacs que `forward-word` ou toute autre fonction d'édition que vous utilisez. (`multiply-by-seven` restera installé jusqu'à ce que vous quittiez Emacs. Pour recharger le code automatiquement chaque fois que vous démarrez Emacs, voir Section 3.5 "Installation

permanente de code", page 3.5.)

Vous pouvez voir l'effet de l'installation de `multiply-by-seven` en évaluant l'échantillon suivant. Placez le curseur après l'expression suivante et tapez `C-x C-e`. Le nombre 21 apparaîtra dans la zone d'écho.

```
(multiply-by-seven 3)
```

Si vous le souhaitez, vous pouvez lire la documentation de la fonction en tapant `C-h f (describe-function)` puis le nom de la fonction, `multiply-by-seven`. Lorsque vous faîtes cela, une fenêtre `*Help*` apparaîtra sur votre écran qui dit :

```
multiply-by-seven is a Lisp function.
(multiply-by-seven NUMBER)
Multiply NUMBER by seven.
```

(Pour revenir à une fenêtre simple sur votre écran, tapez `C-x 1`.)

3.2.1 Changer une définition de fonction

Si vous souhaitez modifier le code dans `multiply-by-seven`, alors réécrivez-le. Pour installer la nouvelle version à la place de l'ancienne, évaluez à nouveau la définition de fonction. Ceci est la façon dont vous modifiez le code dans Emacs. C'est très simple.

À titre d'exemple, vous pouvez modifier la fonction `multiply-by-seven` de sorte qu'elle ajoute le nombre à lui-même sept fois au lieu de multiplier le nombre par sept. Cela produit le même résultat, mais par un chemin différent. En même temps, nous ajouterons un commentaire à ce code ; un commentaire est un texte que l'interprète Lisp ignore, mais qu'un lecteur humain peut trouver utile ou instructif. Le commentaire est «seconde version».

```
(defun multiply-by-seven (number) ; Second version.
 ``Multiply NUMBER by seven.''
 (+ number number number number number number number))
```

Le commentaire suit un point-virgule, ' `;` '. En Lisp, tout sur une ligne qui suit un point-virgule est un commentaire. La fin de ligne est la fin du commentaire. Pour étirer un commentaire sur deux ou plusieurs lignes, chaque ligne doit commencer par un point-virgule.

Voir Section 16.3 "Commencer un fichier `.emacs`", page 187, et Section "Commentaires" dans *The GNU Emacs Lisp Reference Manual*, pour plus de renseignements à propos des commentaires.

Vous pouvez installer cette version de la fonction `multiply-by-seven` en l'évaluant de la même manière que vous avez évalué la première fonction : placez le curseur après la dernière parenthèse et tapez `C-x C-e`.

En résumé, voici comment vous écrivez du code en Emacs Lisp : vous écrivez une fonction ; vous l'installez ; vous la testez ; et ensuite vous faites des corrections ou améliorations et vous l'installez à nouveau.

3.3 Rendre une fonction intéractive

Vous créez une fonction interactive en plaçant une liste qui commence par la forme spéciale `interactive` immédiatement après la documentation. Un utilisateur peut invoquer une fonction interactive en tapant `M-x` et ensuite le nom de la fonction ; ou en tapant la combinaison de touches (raccourci clavier) à laquelle cette fonction est liée, par exemple, en tapant `C-n` pour `next-line` ou `C-x h` pour `mark-whole-buffer`.

Curieusement, lorsque vous appelez une fonction interactive de façon interactive, la valeur renvoyée n'est pas automatiquement affichée dans la zone d'écho. Cela est dû au fait que vous appelez souvent une fonction interactive pour ses effets secondaires, tels que se déplacer vers

l'avant par mot ou par ligne, et non pour la valeur renvoyée. Si la valeur renvoyée était affichée dans la zone d'écho chaque fois que vous tapez un raccourci, cela serait très distrayant.

Tant l'utilisation de la forme spéciale `interactive` que la façon d'afficher une valeur dans la zone d'écho peuvent être illustrées par la création d'une version interactive de `multiply-by-seven`.

Voici le code :

```
(defun multiply-by-seven (number) ; Interactive version.
 ``Multiply NUMBER by seven.''
 (interactive ``p'')
 (message ``The result is %d'' (* 7 number)))
```

Vous pouvez installer ce code en plaçant votre curseur après (la dernière parenthèse) et en tapant `C-x C-e`. Le nom de la fonction apparaîtra dans votre zone d'écho. Ensuite, vous pouvez utiliser ce code en tapant `C-u` et un nombre et ensuite en tapant `M-x multiply-by-seven` et appuer sur `RET`. La phrase `'The result is …'` suivie du produit appraîtra dans la zone d'écho.

De façon plus générale, vous invoquez une fonction de ce genre de l'une des deux manières suivantes :

1. En tapant un argument préfixé qui contient le nombre qui être passé, et ensuite en tapant `M-x` et le nom de la fonction, comme avec `C-u 3 M-x forward-sentence` ; ou,

2. En tapant la touche ou raccourci auquel la fonction est liée, comme avec `C-u 3 M-e`.

Les deux exemples qui viennent d'être mentionnés fonctionnent identiquement pour déplacer le point vers l'avant de trois phrases. (Puisque `multiply-by-seven` n'est liée à aucune touche (ni raccourci), elle ne pouvait pas être utilisée comme un exemple de raccourci de liaison.)

(Voir Section 16.7 “Quelques raccourcis”, page 57, pour apprendre comment lier une commande à une combinaison de touches.)

Un argument préfixé est passé à une fonction interactive en tapant la touche `META` suivi d'un numéro, par exemple, `M-3 M-e`, ou en tapant `C-u` et ensuite le nombre par exemple, `C-u 3 M-e` (si vous tapez `C-u` sans aucun nombre, par défaut ce sera 4).

3.3.1 Une intéractive `multiply-by-seven`

Regardons l'utilisation de la forme spéciale `interactive` et ensuite la fonction `message` dans la version interactive de `multiply-by-seven`. Vous vous souviendrez que la définition de fonction ressemble à cela :

```
(defun multiply-by-seven (number) ; Interactive version.
 ``Multiply NUMBER by seven.''
 (interactive ``p'')
 (message ``The result is %d'' (* 7 number)))
```

Dans cette fonction, l'expression, `(interactive ``p'')`, est une liste à deux éléments. Le ```p''` dit à Emacs de passer l'argument préfixé à la fonction et d'utiliser sa valeur pour l'argument de la fonction.

L'argument sera un nombre. Cela signifie que le symbole `number` sera lié au nombre dans la ligne :

```
(message ``The result is %d'' (* 7 number))
```

Par exemple, si votre argument préfixé est 5, l'interprète Lisp évaluera la ligne comme si elle était :

```
(message ``The result is %d'' (* 7 5))
```

(Si vous lisez ceci dans GNU Emacs, vous pouvez évaluer cette expression vous-même.) Tout d'abord, l'interprète évaluera la liste intérieure, qui est `(* 7 5)`. Cela renvoie une valeur qui est

35. Ensuite, il évaluera la liste extérieure, en passant les valeurs des deuxième éléments et les suivants de la liste à la fonction `message`.

Comme nous l'avons vu, `message` est une fonction Emacs Lisp spécialement conçue pour l'envoi de message d'une ligne pour l'utilisateur. (Voir Section 1.8.5 «La fonction `message`», page 1.8.5.) En résumé, la fonction `message` imprime son premier argument dans la zone d'écho tel quel, sauf pour les occurrences de `'%d'` ou `'%s'` (et les diverses autres %-séquences que nous n'avons pas encore mentionnées). Quand elle voit une séquence de contrôle, la fonction recherche au deuxième ou arguments suivants et imprime la valeur de l'argument à l'emplacement dans la chaîne où la séquence de contrôle est située.

Dans la fonction interactive `multiply-by-seven`, la chaîne de contrôle est `'%d'`, qui requiert un nombre, et la valeur renvoyée par l'évaluation de `(* 7 5)` est le nombre 35. En conséquence, le nombre 35 est imprimé à la place de `'%d'` et le message est `'The result is 35'`.

(Notez que lorsque vous appelez la fonction `multiply-by-seven`, le message est imprimé sans guillemets, mais quand vous appelez `message`, le texte est imprimé avec guillemets. C'est parce que la valeur renvoyée par `message` est ce qui appraît dans la zone d'écho quand vous évaluez une expression dont le premier élément est `message` ; mais lorsqu'il est incorporé dans une fonction, `message` imprime le texte comme un effet secondaire sans guillemets.)

3.4 Différentes options pour `interactive`

Dans l'exemple, `multiply-by-seven` utilisait `''p''` comme un argument pour `interactive`. Cet argument disait à Emacs d'interpréter votre frappe soit `C-u` suivi d'un nombre soit `META` suivi d'un nombre comme une commande à passer ce nombre à la fonction comme son argument. Emacs a plus de vingt caractères prédéfinis pour l'utilisation avec `interactive`. Dans presque tous les cas, une de ces options vous permettra de passer la bonne information interactivement à la fonction. (Voir Section «Caractère de code pour `interactive`» dans *The GNU Emacs Lisp Reference Manual*.)

Considérez la fonction `zap-to-char`. Son expression interactive est

```
(interactive ''p\ nZap to char : '')
```

La première partie de l'argument de `interactive` est `'p'`, avec ce dont vous êtes déjà familier. Cet argument dit à Emacs d'interpréter un «préfixe», comme un nombre qui doit être passé à la fonction. Vous pouvez spécifier un préfixe soit en tapant `C-u` suivi d'un nombre soit en tapant `META` suivi par un nombre. Le préfixe est le nombre de caractères spécifiés. Ainsi, si votre préfixe est trois et le caractère spécifié est `'x'`, alors vous effacerez tout le texte jusqu'à et y compris la troisième occurence de `'x'`. Si vous n'initialisez pas un préfixe, alors vous effacerez tout le texte jusqu'à et y compris le caractère spécifié, mais pas plus.

Le `'c'` indique à la fonction le nom du caractère qu'elle doit supprimer.

Plus formellement, une fonction avec plusieurs arguments peut avoir des informations passées à chacun des argument en ajoutant des parties à la chaîne qui suit `interactive`. Lorsque vous faîtes ça, l'information est passée à chaque argument dans le même ordre qu'elle est spécifiée dans la liste `interactive`. Dans la chaîne, chaque partie est séparée de la suivante par un `'\'`, qui est une nouvelle ligne. Par exemple, vous pouvez suivre `'p'` avec un `'\ n'` et un `'cZap to char : '`. Cela incite Emacs à passer la valeur de l'argument préfixé (s'il y en a un) et le caractère.

Dans ce cas, la définition ressemble à la suivante, où `arg` et `char` sont les symbole auxquels `interactive` lie l'argument préfixé et le caractère spécifié :

```
(defun name-of-function (arg char)
```

```
''documentation…''
(interactive ''p\ ncZap to char; '')
body-of-function…)
```

(L'espace après les deux points dans l'invite lui donne une meilleure apparence lorsque vous êtes invité. Voir Section 5.1 «La définition de `copy-to-buffer`», page 35, par exemple.)

Quand une fonction ne prend pas d'arguments, `interactive` n'en nécessite aucun. Une telle fonction contient l'expression simple `(interactive)`. La fonction `mark-whole-buffer` est comme ça.

Alternativement, si les lettres-codes ne sont pas bonnes pour votre application, vous pouvez passer vos propres arguments à `interactive` comme une liste.

Voir Section 4.4 «La définition de `append-to-buffer`», page 33, par exemple. Voir Section «Utilisation de `Interactive`» dans *The GNU Emacs Lisp Reference Manual*, pour une explication plus complète à propos de cette technique.

3.5 Installer du code de façon permanente

3.6 `let`

3.6.1 Les parties d'une expression `let`

3.6.2 Échantillon d'expression `let`

3.6.3 Variables non initialisées dans une instruction `let`

3.7 La forme spéciale `if`

3.7.1 La fonction `type-of-animal` en détail

3.8 Les expressions If-then-else

3.9 Vérité et fausseté dans Emacs Lisp

3.10 `save-excursion`

3.10.1 Modèle pour une expression `save-excursion`

3.11 Rappels

3.12 Exercices

Chapitre 4

Quelques fonctions liées aux tampons

Chapitre 5

Quelques fonctions complexes

Chapitre 6

Réduction et agrandissement

6.1 La forme spéciale de `save-restriction`

6.2 `what-line`

6.3 Exercice avec réduction

Chapitre 7

`car`, `cdr`, `cons` : Fonctions Fondamentales

7.1 `car` et `cdr`

7.2 `cons`

7.2.1 Trouver la longueur d'une liste : `length`

7.3 `nthcdr`

7.4 `nth`

7.5 `setcar`

7.6 `setcdr`

7.7 Exercice

Chapitre 8

Couper et stocker du texte

Chapitre 9

Comment les listes sont implémentées

9.1 Symboles en tant que commode

9.2 Exercice

Chapitre 10

Récupération du texte

10.1 Kill ring overview

10.2 La variable `kill-ring-yank-pointer`

10.3 Exercices avec `yank` et `nthcdr`

Chapitre 11

Boucles et récursivité

11.1 `while`

11.1.1 Une boucle while et une liste

11.1.2 Un exemple : `print-elements-of-list`

11.1.3 Une boucle avec un compteur d'incrémentation

11.1.4 Une boucle avec un compteur de décrémentation

11.2 Économisez votre temps : `dolist` et `dotimes`

11.3 Récursivité

11.3.1 Construction de robots : extension de la métaphore

11.3.2 Les parties d'une définition récursive

11.3.3 Récursivité avec une liste

11.3.4 Récursivité au lieu d'itérations

11.3.5 Exemple de récursivité utilisant `cond`

11.3.6 Masques récursifs

11.3.7 Récursivité sans sursis

11.3.8 Aucune solution de report

11.4 Exercice de boucles

Chapitre 12

Recherches d'expressions régulières

Chapitre 13

Comptage : répétition et expressions régulières

13.1 La fonction `count-words-example`

13.1.1 Les bogues des espaces dans `count-words-example`

13.2 Comptage des mots exclusivement

13.3 Exercice : compter la ponctuation

Chapitre 14

Comptage des mots dans une `defun`

Chapitre 15

Préparation d'un graphe

Chapitre 16

Votre fichier `.emacs`

Chapitre 17

Débogage

17.1 `debug`

17.2 `debug-on-entry`

17.3 `debug-on-quit` et (`debug`)

17.4 Le `edebug` Source Level Debugger

17.5 Exercices de débogage

Chapitre 18

Conclusion

Annexe A

La fonction `the-the`

Annexe B

Manipulation du Kill Ring

B.1 La fonction `current-kill`

B.2 `yank`

B.3 `yank-pop`

B.4 Le fichier `ring.el`

Annexe C

Un graphe avec des axes étiquetés

C.1 La Varlist `print-graph`

C.2 La fonction `print-Y-axis`

C.2.1 Calculer un reste

C.2.2 Construire un élément de l'axe des ordonnées

C.2.3 Créez une colonne de l'axe des ordonnées

C.2.4 La version finale de Not Quit `print-Y-axis`

C.3 La fonction `print-X-axis`

C.3.1 Marques tic de l'axe des abscisses

C.4 Tracer le graphe complet

C.4.1 Tester `print-graph`

C.4.2 Nunméros de graphe des mots et symboles

C.4.3 Une expression lambda : utilité de l'anonymat

C.4.4 La fonction `mapcar`

C.4.5 Un autre bogue ...plus insidieux

C.4.6 Le graphe tracé

Annexe D

Logiciel libre et manuels libres

Annexe E

Licence de documentation libre GNU