Langages Enchâssés

Robert Strandh
professeur à l'Université Bordeaux 1

Irène Durand
maître de conférences à l'Université Bordeaux 1

Juin 2004

Table des matières

Chapitre 1

Introduction

1.1 Public

Ce livre est destiné aux étudiants de Master Informatique et aux élèves des Écoles d'ingénieurs en Informatique qui s'orientent vers une formation en Génie Logiciel.

Sa lecture nécessite la connaissance préalable du langage Common Lisp [2] [4] [3] [6] [7] ainsi que des notions de Génie Logiciel.

1.2 Objectifs

Les langages enchâssés constituent une technique très puissante pour la construction de langages dédiés à un domaine d'application.

Le premier objectif du livre est de convaincre le lecteur de l'utilité de langages dédiés pour la construction de logiciels de taille importante et de qualité.

Le deuxième objectif est de montrer que pour implémenter un langage dédié, la solution qui consiste à développer un langage enchâssé est particulièrement adaptée.

Finalement, nous souhaitons montrer au lecteur comment se servir du système de macros Common Lisp pour l'implémentation d'un tel langage.

1.3 Choix du langage

Pour ce livre, nous avons choisi le langage Common Lisp. C'est un langage de la famille Lisp. Lisp existe depuis 1959, initialement développé par John McCarthy et son groupe de recherche au Massachusetts Institute of Technology (MIT).

Pour se prêter à l'écriture de langages enchâssés, un langage de programmation doit disposer d'un système de macros très puissant. Les macros basées sur un mécanisme de substitution de texte comme celles proposées par le préprocesseur du langage C ou le macro-processeur M4 ne sont pas assez évoluées même si M4 est légèrement supérieur au préprocesseur C, car capable d'évaluer des expressions arithmétiques. Le langage Common Lisp est actuellement le seul langage général à proposer un système de macros sophistiqué permettant de manipuler le langage dans son ensemble.

En principe, rien n'empêche d'étendre M4 de manière à ce qu'il contienne un langage de programmation complet. Mais cela ne suffit pas pour obtenir un système de macros aussi puissant que celui de Common Lisp. L'autre qualité unique de Common Lisp est que le format interne (sous forme de listes) d'un programme, utilisé par le compilateur mais aussi par le système des macros, est normalisé. En fait, un bon programmeur Common Lisp ne pense pas à son programme comme un texte, mais il traduit, tel le compilateur, ce texte en format interne. Le système des macros travaille sur cette représentation interne. Il reçoit des expressions en représentation interne en entrée et en produit de nouvelles, également en représentation interne.

Un langage de programmation est bien adapté à l'écriture de macros, s'il est capable de représenter ses propres programmes comme des données. La famille des langages Lisp est unique en ce sens. On peut donc dire que tout langage adapté à l'écriture de macros, voire capable d'être un langage hôte pour un système de macros puissant, et donc pour des langages enchâssés, sera forcément assez proche de la famille Lisp.

Aujourd'hui, le seul représentant raisonnable de la famille Lisp est le langage Common Lisp. Sa définition est le résultat de plusieurs décennies de réflexion et d'expérimentation. Il est donc juste de dire que nous n'avions pas de véritable choix de langage pour ce livre, puisque Common Lisp est le seul langage raisonnable, voire possible.

1.4 Conventions

Nous utilisons une police comme `celle-ci` pour toute interaction avec le système Common Lisp et tout code de programme. La même police `soulignée` est utilisée pour les messages du système Common Lisp vers l'utilisateur.

Une police comme *celle-ci* est utilisée pour les valeurs Common Lisp en représentation interne. Par exemple, le résultat de l'entrée de l'expression `234` est le nombre *234*.

1.5 Organisation du livre

Ce livre est organisé en trois parties. La première partie présente les aspects du génie logiciel qui sont à l'origine de la création des langages enchâssés. Dans cette première partie, nous traitons aussi en partie l'histoire de certains aspects du génie logiciel utiles pour le reste du livre.

La deuxième partie du livre est entièrement consacrée à l'écriture de macros Common Lisp. Le système de macros Common Lisp est très riche (en fait, le langage Common Lisp entier est à la disposition du programmeur de macros). En conséquence des idiomes se sont développés dans la communauté et doivent obligatoirement être respectés par tout programmeur expert. Ces idiomes sont traités en détails dans cette partie.

La troisième partie contient des exemples de langages enchâssés. En particulier, nous décrivons un langage similaire à PostScript destiné à la génération de pages imprimables contenant texte et graphiques. Nous suivons l'évolution de ce petit langage d'abord implémenté sous la forme d'un interprète, puis sous la forme d'un véritable langage enchâssé.

Première partie

Aspects du génie logiciel

Chapitre 2

Programmation ascendante ou descendante

On distingue deux méthodes principales de conception de programmes. La première, très en vogue dans les années 1970 est connue sous le nom de *programmation structurée* encore appelée *programmation descendante* . La deuxième, la *programmation ascendante*, est une réaction à la première et peut être considérée comme la base de la programmation par [5] objets.

Dans ce chapitre, nous expliquons la différence entre les deux.

2.1 Programmation descendante

L'idée de la programmation descendante est de commencer par une description de très haut niveau (une spécification) du système cible. L'essence même de la programmation descendante est certainement la méthode proposée par le créateur du langage de programmation Pascal, Nicklaus Wirth, et nommée « raffinages successifs » (en anglais : stepwise refinement).

À chaque étape de cette méthode, une description d'un niveau non élémentaire est raffinée en plusieurs descriptions de niveau inférieur. Le processus se termine lorsque chaque description est d'un niveau suffisamment bas pour être directement traduite en instructions du langage de programmation. À ce stade, il suffit de coder la spécification. Cette méthode ayant été appliquée à l'ensemble des sous-descriptions, toute la spécification a été traduite

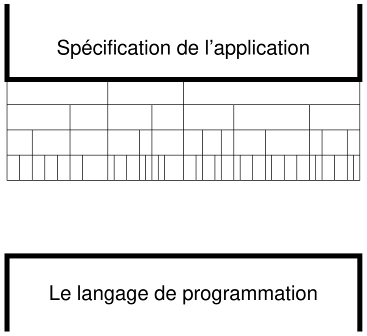

Fig. 2.1 – La programmation descendante

en code et le système est terminé. Ce processus est illustré par la figure 2.1.

Les *modules* qui résultent de la programmation descendante correspondent souvent à une *fonctionnalité* du système. Il est raisonnable de se demander ce que *fait* un tel module.

Ce type de programmation a des inconvénients majeurs que nous explicitons ci-après.

D'abord, les modules qui en résultent sont très spécialisés et donc peu réutilisables. C'est simplement le résultat immédiat de la méthode. Chaque module est le résultat du raffinement d'un module de plus haut niveau. Dans l'utilisation professionnelle de cette méthode, il est courant de répartir la réalisation des descriptions de haut niveau entre plusieurs développeurs. En raffinant leur description, il se peut que deux développeurs soient amenés à définir le même module. Ce module sera développé plusieurs fois indépendamment, probablement de façon légèrement différente à chaque fois.

Ensuite, et ceci est encore plus important, la programmation descendante entraîne la nécessité définir de manière détaillée les fonctionnalités du système final (la spécification détaillée du système) très tôt dans le projet. Même s'il peut paraître normal de savoir ce que doit faire le système avant de commencer le développement, en réalité, ce fait est à l'origine de beaucoup d'échecs du développement de grands systèmes informatiques.

Le problème principal est qu'il est quasiment impossible d'exiger de la part du client une spécification détaillée du système final *a priori*. En fait la spécification ne peut jamais être complètement détaillée tant que le système n'est pas développé. La spécification détaillée serait en fait pratiquement équivalente au code final, ce que le client veut justement qu'on réalise pour lui. De plus il est très difficile de savoir *a priori* ce dont on aura besoin. La plupart du temps, la spécification évolue considérablement pendant la période de développement du système. Avec la programmation descendante, on doit décider prématurément de ce que doit faire le système et tout développement dépend de cette spécification initiale. Une fois le développement commencé, il est difficile, voire impossible de modifier les fonctionnalités du système sans invalider l'effort de développement effectué jusqu'alors.

On pratique malheureusement toujours la programmation descendante, avec des résultats assez catastrophiques. Selon des études de l'armée américaine, sur 10 des systèmes développés sur mesure, 9 environ ne sont jamais mis en service. Entre l'établissement de la spécification et la livraison du système, les souhaits du client ont tellement évolué que le système livré n'est plus d'actualité. Parmi les systèmes mis en service, une grande majorité l'est seulement après des révisions profondes et coûteuses nécessaires dues à l'adaptation du système aux nouveaux souhaits du client.

2.2 Programmation ascendante

La programmation ascendante est une réaction à la programmation descendante. Avec ce type de programmation, on ne commence pas par une description complète de ce que doit faire le système. Cette méthode n'a donc pas l'inconvénient de la programmation descendante de devoir déterminer trop tôt les fonctionnalités du système.

La plupart du temps, les modules de la programmation ascendante ne correspondent pas à des fonctionnalités, mais à des *types de données* ou *types abstraits* du système final. On s'aperçoit que ces types sont plus stables que les fonctionnalités du système. Un système de réservation de billets d'avions sera toujours amené à manipuler des représentations de vols, d'aéroports, de passagers, de classes, etc., indépendamment des fonctionnalités finales du système.

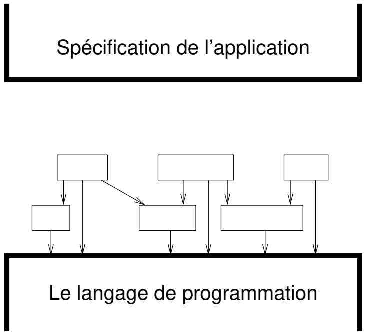

FIG. 2.2 – La programmation ascendante

Avec la programmation ascendante, le développeur construit des types abstraits, d'abord à partir du langage de programmation « nu », puis en réutilisant les types abstraits (modules) déjà construits. Le résultat est une collection de modules assez indépendants. Ces modules sont relativement faciles à tester au fur et à mesure car ils sont 'écrits en utilisant le langage de programmation et donc exécutables. Le degré de réutilisation des modules est assez élevé, car lorsqu'un module est développé, il est naturel de regarder si un module existant peut être utilisé pour effectuer une partie du travail. Ce processus est illustré par la figure 2.2.

La programmation ascendante a évolué et s'utilise aujourd'hui à travers la *programmation par objets* laquelle offre en plus la possibilité d'exprimer des types abstraits par la spécialisation de types abstraits existants.

Une façon de voir les modules qui résultent de la programmation ascendante est qu'ils représentent une extension du langage de programmation de base pour l'écriture d'une famille d'applications. Une collection de modules pour la manipulation d'avions, de passagers, de billets, etc, représente donc une extension du langage de programmation pour l'écriture de systèmes de réservation de billets d'avions. Les modules ne sont pas spécialisés pour un seul système, mais adaptés à une famille de systèmes (ou à un *domaine* d'application) car ils ont été réalisés sans tenir compte de ce que doit faire le système final en détail.

Il est évident que certains modules doivent être spécialisés pour le système final de manière à réaliser les fonctionnalités souhaitées. Or il est possible que la plupart des modules soient plus généraux et qu'ils soient réutilisables dans d'autres systèmes similaires.

Chapitre 3

Langages dédiés

Dans la section 2.2, nous avons vu qu'une collection de modules peut être vue comme une extension d'un langage de programmation, spécialisée pour un domaine d'application.

Dans ce chapitre, nous reprenons cette idée et la poussons à l'extrême. Nous souhaitons en effet disposer d'un langage de programmation contenant déjà une grande partie des fonctionnalités nécessaires pour un domaine d'application. Il est clair que s'il est possible pour une équipe de développement de se procurer un tel langage, son travail est presque terminé. En réalité, ce sera le travail de l'équipe même de réaliser le langage. Nous allons voir comment on peut envisager la réalisation d'un tel langage.

Nous allons souvent parler de la notion de *langage hôte* . Il s'agit du langage utilisé pour implémenter le langage dédié. Dans certains cas, le langage hôte ne sera pas directement visible par l'utilisateur du langage dédié. Dans d'autres cas, l'application sera un mélange entre le langage dédié et le langage hôte.

3.1 Qualités d'un langage dédié

La différence essentielle entre une collection de modules dans un langage général (voir section 2.1) et un langage dédié est l'existence dans ce dernier de *syntaxe* spécialisée. Dans cette section, nous allons développer cette notion.

Il est reconnu qu'une des bases de la programmation est la construction systématique d'*abstractions*. Trois types d'abstractions sont identifiés à ce jour :

– l'abstraction fonctionnelle. Il s'agit de la construction de *procédures* et de *fonctions* pour capturer et nommer les idiomes ainsi exprimés ;
– l'abstraction de données. C'est la construction de *types abstraits* dont l'application a besoin ;
– l'abstraction syntaxique. Cette abstraction crée de nouvelles constructions syntaxiques, ce qui permet une expression plus claire et plus dense de certaines constructions de l'application et du domaine de l'application.

La plupart des langages de programmation existant ne permet que les deux premiers types d'abstractions.

Pour un langage dédié, il est crucial de disposer de constructions syntaxiques spécifiques au domaine d'application. Il est toujours *possible* d'exprimer de telles constructions par le biais de fonctions et de procédures, mais cela risque fort de devenir assez lourd. De plus, dans ce cas, le compilateur ne pourra pas détecter des fautes qui seraient pourtant faciles à détecter dans le cas d'une construction syntaxique.

Le langage graphique OpenGL est un bon exemple de langage dédié implémenté sans extensions syntaxiques et entièrement sous la forme d'une collection de fonctions en langage C.

Un autre exemple est le langage PostScript qui manipule son environnement graphique grâce à des fonctions permettant de sauver et restaurer l'environnement courant. À ce propos, une erreur fréquente en PostScript est de faire déborder la pile d'environnements car il est facile d'oublier de restaurer l'environnement. Une construction syntaxique, sous forme de parenthèses équilibrées permettrait au compilateur de détecter ce défaut.

Cependant l'exemple le plus flagrant est peut-être la construction d'interfaces graphiques réalisées presque exclusivement par des suites d'appels à des fonctions de création de boutons, de fenêtres, etc. Un peu de syntaxe permettrait à ces programmes d'être beaucoup plus lisibles.

Les langages de programmation modernes contiennent pourtant des constructions syntaxiques, preuves que ces constructions sont jugées utiles. En voici quelques exemples :

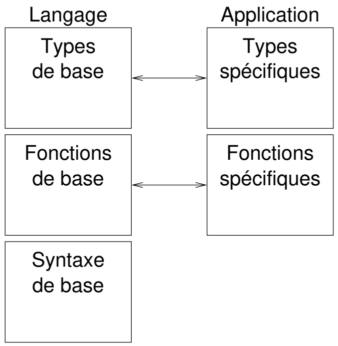

FIG. 3.1 – Abstractions dans les langages traditionnels

- le passage de paramètres. À part dans certains langages comme Forth ou PostScript, on ne manipule pas explicitement le mécanisme de passage de paramètres à des sous-programmes. On utilise une syntaxe permettant au compilateur de vérifier certains aspects de ce passage comme le nombre et le type des paramètres ;
- la notion de *bloc*. Pour la création et la destruction de variables locales, on utilise une construction syntaxique avec des parenthèses équilibrées. En langage C, il s'agit d'accolades, en Algol des mots-clés **begin** et **end**, etc. On peut difficilement imaginer un langage nécessitant un appel de fonction pour la création d'une variable locale et un autre appel pour la détruire ;
- la création de classes et de méthodes. C'est encore de la syntaxe spécifique qui permet au compilateur d'effectuer certaines vérifications qui ne seraient pas possible si la création de classes se faisait uniquement par des appels de fonctions. La syntaxe permet surtout au programmeur d'exprimer de manière plus compacte les constructions souhaitées.

Il y a une lacune flagrante dans la plupart des langages de programmation existants. Ils donnent au programmeur la possibilité de créer ses propres fonctions, procédures et types de données, mais pas de nouvelles constructions syntaxiques. Cette situation est illustrée par la figure 3.1.

Dans la suite, nous allons voir que la possibilité offerte au programmeur de

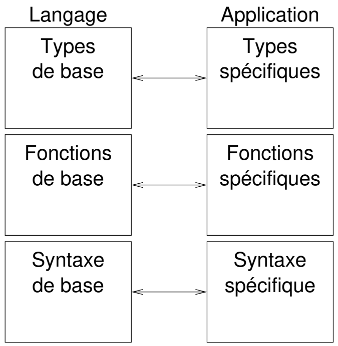

FIG. 3.2 – Abstractions en Common Lisp

créer ses propres constructions syntaxiques est cruciale pour l'implémentation de langages dédiés.

Le langage Common Lisp est un des rares langages permettant ce type de constructions et donc adapté à la création de langages enchâssés. Le fait que Common Lisp admette les trois types d'abstractions est illustré par la figure 3.1.

3.2 Techniques d'implémentation d'un langage dédié

Il y a plusieurs façons d'envisager l'implémentation d'un langage dédié. Comme expliqué dans la section précédente, le but est de fabriquer des constructions syntaxiques spécifiques au domaine cible. Une implémentation classique utilisant procédures, fonctions, classes, etc., d'un langage de programmation traditionnel ne le permet pas.

3.3 Préprocesseur

La première possibilité est de construire un *préprocesseur*. C'est la technique utilisée pour le langage Pro*C lequel permet de mélanger du C et des instructions SQL pour accéder à une base de donnée relationnelle.

Le programme passe d'abord par un préprocesseur qui traduit les instructions SQL en constructions plus élémentaires en langage C, souvent des appels de fonctions pour ouvrir, fermer la base et y accéder. Puis le programme qui en résulte est compilé par le compilateur C habituel.

Ce type d'implémentation permet au programmeur d'ignorer les détails de l'implémentation du langage C et de se concentrer sur l'expression des constructions du langage dédié (ici SQL) sous la forme d'instructions du langage hôte (ici le C).

Un avantage considérable de cette technique est la possibilité de réutiliser *en masse* les fonctionnalités de base du langage hôte. Ainsi, le créateur du préprocesseur est déchargé de l'implémentation des fonctionnalités de base comme l'arithmétique, l'appel de fonction, les boucles simples, etc. Il suffit de ne pas traiter ces constructions et de les faire passer au compilateur du langage hôte.

Par contre, il y a plusieurs inconvénients. Il est courant que le programmeur de Pro*C reçoive des messages d'erreurs du compilateur C faisant référence à du code généré par le préprocesseur. Ces messages sont souvent incompréhensibles, car le programmeur ne connaît pas les détails du fonctionnement du préprocesseur. Il est bien sûr possible d'éviter ces messages avec un préprocesseur capable de comprendre à la fois le langage hôte entier et le langage dédié. De cette manière, on est sûr de générer du code sans erreurs syntaxiques. Mais en pratique, il n'est pas possible que le préprocesseur ait une connaissance aussi profonde du langage hôte.

Un autre problème lié à cette technique se pose dès que l'on souhaite mélanger deux ou plusieurs langages dédiés. On est forcé de choisir un ordre d'application des préprocesseurs. Or le premier préprocesseur ne reconnaît pas forcément le langage du deuxième et vice-versa. De plus, il est impossible pour un préprocesseur de profiter de l'existence de l'autre et de générer du code contenant l'autre langage.

3.4 Interprète

Une autre façon d'implémenter un langage dédié est d'écrire un interprète pour ce langage. C'est le choix de certains langages dits « de scriptage » comme Tcl, Python, Perl, etc.

L'avantage d'une telle implémentation est la maîtrise totale de toutes les constructions et donc la possibilité pour l'interprète d'émettre des message pertinents. L'addition de nouvelles constructions se fait par la modification de l'interprète.

Dans ce cas encore, il y a de nombreux inconvénients. D'abord il n'est pas possible de profiter des constructions de base du langage hôte, (celui utilisé pour implémenter l'interprète). Il s'agit d'un langage complètement nouveau. Il est donc nécessaire d'écrire du code explicite dans l'interprète pour reconnaître les constructions arithmétiques, les boucles, etc. L'implémentation du langage dédié nécessite alors des connaissances assez importantes sur l'implémentation de langages de programmation.

L'écriture d'un interprète est considérablement plus difficile que celle d'un préprocesseur ; elle prend par conséquent beaucoup plus de temps. De plus, un investissement insuffisant dans le traitement des erreurs peut facilement rendre l'interprète inutilisable par d'autres personnes que ses créateurs.

Finalement, un interprète est plus lent à l'exécution que le code généré par un préprocesseur puis compilé par un compilateur natif pour le langage hôte. Beaucoup d'applications nécessitent une exécution rapide. L'utilisation d'un langage dédié implémenté sous la forme d'un interprète peut nécessiter de nombreuses modifications de l'interprète dans le but d'augmenter la performance de certaines constructions, voire la possibilité de mélanger le langage dédié et le langage hôte ce qui laisse le choix entre la performance du langage hôte et les constructions spécifiques du langage dédié mais pas les deux simultanément.

3.5 Compilateur

La lenteur d'un interprète peut être éliminée si l'on implémente le langage dédié par un compilateur plutôt que par un interprète. Le compilateur peut générer du code en langage hôte afin d'éviter la dépendance avec un processeur particulier. Mais cela ne résout pas le problème de la duplication des constructions de base comme l'arithmétique. Et en plus, on rajoute encore des inconvénients, surtout pour les personnes qui doivent implémenter ce compilateur. Un compilateur est considérablement plus difficile à écrire qu'un interprète. La performance n'est pas automatique, il faut être expérimenté dans les diverses techniques de compilation.

3.6 Définition de langage enchâssé

Après la description de plusieurs techniques d'implémentation de langages dédiés, nous sommes finalement en mesure de définir la notion de *langage enchâssé*.

Il s'agit d'une technique d'implémentation d'un langage dédié basée sur l'utilisation systématique d'extensions syntaxiques du langage hôte.

Le langage hôte est augmenté de procédures, de fonctions et de types spécifiques au langage dédié, mais aussi et en particulier de constructions syntaxiques spécifiques.

Cette manière d'implémenter un langage dédié a plusieurs avantages :

– le langage dédié est compilé si le langage hôte l'est ;
– elle permet l'importation *en masse* de constructions du langage hôte (arithmétique, boucles, etc) ;
– il est possible de mélanger plusieurs langages dédiés sans problèmes de compatibilité ;
– l'implémentation du langage dédié est aussi simple que l'implémentation d'un préprocesseur.

Nous voyons que cette technique a tous les avantages des autres mais aucun des inconvénients.

Chapitre 4

Caractéristiques du langage hôte

Dans ce chapitre, nous allons examiner les caractéristiques nécessaires d'un langage hôte pour qu'il admette un langage enchâssé.

Dans le chapitre 3, nous avons analysé les trois types d'abstractions de base pour la construction d'un langage dédié. Nous avons particulièrement insisté sur la nécessité de l'existence d'un mécanisme d'abstraction syntaxique. Mais qu'est-ce que l'abstraction syntaxique ? De la même façon que l'abstraction fonctionnelle permet la création de fonctions à partir de fonctions existantes et l'abstraction de données permet la création de types à partir de types existants, l'abstraction syntaxique permet la création de syntaxe à partir de syntaxe existante. Il s'agit donc de la traduction d'une construction syntaxique en d'autres constructions syntaxiques.

Une autre façon de dire la même chose est que l'abstraction syntaxique est la même chose qu'un mécanisme de macros et de macro-substitution.

4.1 Les macros basées sur la substitution de texte

Il y a plusieurs types de macros. En particulier, les macros basées sur la substitution de texte, comme les macros du langage C, sont inexploitables du point de vue d'un langage enchâssé. La raison est que les macros de ce type ne sont pas capables de respecter la structure de base du langage.

Le système de macros du langage C n'exige pas que les arguments d'une macro soient des expressions, instructions ou autres éléments du langage, et donc le résultat de la macro-expansion ne l'est pas nécessairement non plus. Les macros C nécessitent alors une connaissance assez détaillée du fonctionnement interne de la macro, en particulier de son comportement dans différents types de contextes.

Voici un exemple typique :

```
#define max(x, y) x > y ? x : y
```

Cette macro semble calculer le maximum entre ses deux arguments. C'est le cas dans un contexte comme celui-ci :

```
a = max(b, c);
```

mais seulement parce que l'affectation est moins prioritaire que la comparaison et l'expression conditionnelle. Dans le contexte suivant :

```
a = d * max(b, c);
```

il y a un problème, car la multiplication est plus prioritaire que la comparaison. L'expansion de la macro donne :

```
a = d * b > c ? b : c;
```

ce qui compare le produit de `d` et de `b` avec `c` et non simplement `b` avec `c` comme l'indique l'expression avant expansion. Le programmeur C expert connaît bien ces pièges potentiels et rajoute un grand nombre de parenthèses pour éviter ce type de problème. On obtient alors plutôt la définition suivante :

```
#define max(x, y) ((x) > (y) ? (x) : (y))
```

Mais même avec ces précautions, il est impossible d'éviter certains problèmes, comme par exemple l'évaluation multiple. Dans l'exemple suivant :

```
a = max(b++, c++)
```

on peut croire que l'affectation donne la valeur la plus grande des deux avant l'incrémentation, mais ce n'est pas le cas comme nous le montre la macro-expansion :

```
a = ((b++) > (c++) ? (b++) : (c++))
```

Ici, on voit que le plus grand des deux sera incrémenté deux fois et le résultat de l'affectation sera la valeur après une incrémentation. Il n'y a aucune manière d'éviter ce type de problèmes avec des macros basées sur la substitution de texte.

C'est sans doute ce type de phénomènes non souhaitables des macros du langage C qui a incité les créateurs du langage Java à les exclure de ce langage. Ceci est dommage car, alors que les macros C ne sont pas parfaites, elle permettent néanmoins certaines extensions syntaxiques impossibles en Java.

Dans la section suivante, nous allons voir comment les macros Common Lisp réussissent à éviter tous les problèmes indiqués dans cette section.

4.2 Les macros Common Lisp

La syntaxe simple et uniforme joue un rôle crucial dans l'élaboration d'un système de macros évitant les difficultés d'un système basé sur la substitution de texte.

En fait, en Common Lisp, la représentation externe d'un programme sous forme de suite de caractères est une traduction immédiate (effectuée par la fonction `print`) de la représentation de l'*arbre syntaxique* du programme. La correspondance entre la représentation externe et la représentation interne est même exigée par la norme.

Cette correspondance est si directe que le programmeur Common Lisp fait automatiquement la traduction. Par exemple, lorsqu'il voit la suite des caractères :

```
(+ 3 4)
```


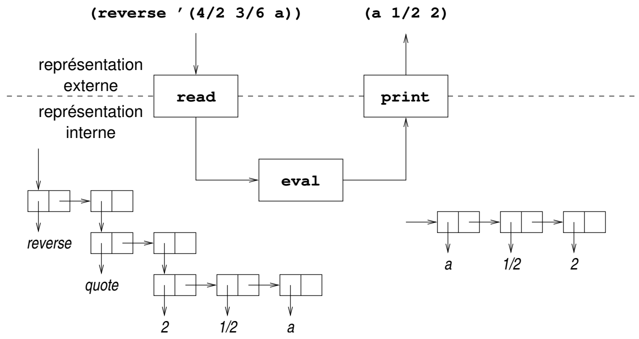

FIG. 4.1 – La boucle `read-eval-print`

il ne pense pas « parenthèse ouvrante suivie du caractère '+', etc », mais « une liste de trois éléments, un symbole et deux entiers ». En fait, le langage Common Lisp est défini par une boucle d'interaction, ou boucle `read-eval-print` (voir figure 4.1). Une expression est convertie par `read` de sa représentation externe en sa représentation interne, puis évaluée par `eval` et sa valeur est finalement convertie par `print` en représentation externe pour affichage.

C'est cette correspondance entre la représentation externe et la représentation interne qui permet le mécanisme des macros Common Lisp. Les macros Common Lisp ne remplacent pas du texte par du texte, mais *un arbre syntaxique par un autre arbre syntaxique*. Le programmeur serait totalement incapable d'écrire une macro s'il ne connaissait pas la forme des arbres syntaxiques associés aux expressions souhaitées.

La plupart des langages de programmation existants ne normalise pas la forme de l'arbre syntaxique. Il est donc difficile, voire impossible, d'imaginer un système de macros viable pour ces langages.

Le mécanisme de macro-substitution Common Lisp est donc basé sur la transformation d'un arbre syntaxique en un autre arbre syntaxique. Mais comment sont exprimées ces transformations ? Il serait bien sûr possible d'inventer un formalisme spécifique pour ces transformations. Mais là encore, Common Lisp a un avantage. Les arbres syntaxiques en Common Lisp sont exprimés sous la forme de *structures de listes* et la liste est l'un des types de données le mieux supporté par le langage. Autrement dit, les programmes Common Lisp, sous la forme d'arbres syntaxiques, sont représentés par des listes manipulables par les fonctions du langage même. Il est donc naturel de proposer que ce soit un programme Common Lisp qui soit responsable de la transformation d'un arbre syntaxique en un autre arbre syntaxique. C'est exactement le cas. Une macro Common Lisp est simplement un programme Common Lisp qui prend des expressions Common Lisp en entrée et qui produit une expression Common Lisp en sortie.

Pour envisager un langage enchâssé dans un autre langage que Lisp, il faut que ce langage ait les caractéristiques suivantes :

- syntaxe régulière et simple,
- syntaxe dont la représentation interne soit évidente,
- la représentation interne de l'arbre syntaxique doit correspondre à un type de donnée de base du langage même,.

Un tel langage sera forcément très proche de Lisp.

Deuxième partie

Écrire des macros en Common Lisp

Chapitre 5

Le mécanisme de macro-substitution

Une macro est une fonction qui accepte du code Common Lisp sous forme d'expressions passées en argument et qui renvoie comme valeur une nouvelle expression. Le compilateur, avant de compiler un appel macro (une expression commençant par le nom d'une macro), appelle une fonction qui réalise l'*expansion* de l'appel.

5.1 Première macro par l'exemple

Plaçons nous dans le cadre d'une application d'animation d'images. Si les pixels produits par le processeur graphique sont directement écrits ligne par ligne dans la mémoire de la carte graphique on risque d'obtenir des scintillements désagréables. La technique du *double buffer* (en anglais : double buffering) permet de résoudre ce problème et d'augmenter l'impression de fluidité des animations à l'écran. Elle consiste à utiliser un buffer auxiliaire ou *pixmap* dans lequel le processeur graphique écrit les pixels. Quand l'image est complète l'ensemble du buffer est copié dans la zone la mémoire de la carte graphique qui s'affiche à l'écran. L'avantage est que l'écran, à chaque rafraîchissement, dispose toujours d'une image entière, c'est-à-dire totalement construite et et non en cours d'élaboration.

Un pixmap prend beaucoup d'espace mémoire et doit être libéré dès qu'il

n'est plus utilisé.

Dans l'application, on trouve donc des suites d'instructions de ce type :

```
(setf (pixmap screen) (allocate-pixmap))
instruction-1
...
instruction-n
(copy-from-pixmap screen)
(setf (pixmap screen) nil)
```

L'idée est d'allouer un pixmap temporaire et de le libérer dès que possible. En fait, cet exemple pose un problème assez grave. Si l'un des *instruction-i* provoque une erreur, l'instruction qui libère l'espace occupé par le pixmap (`(setf (pixmap screen) nil)`) ne sera pas exécutée. L'espace mémoire occupé par le pixmap ne sera pas récupéré par le rammasse-miette avant la prochaine affectation. Pour éviter ce problème, il faut écrire :

```
(setf (pixmap screen) (allocate-pixmap))
(unwind-protect
    (progn
      instruction-1
      ...
      instruction-n
      (copy-from-pixmap screen))
  (setf (pixmap screen) nil))
```

Ici l'expression *unwind-proctect* garantit que la dernière affectation

```
(setf (pixmap screen) nil)
```

sera effectuée, même en cas d'erreur. La suite d'expressions ci-dessus sera sans doute répétée un grand nombre de fois dans l'application. Elle devient alors un *idiome* de l'application. Chaque fois que l'application a besoin d'afficher une image, le programmeur doit entourer ses calculs avec les expressions ci-dessus. Cela pose des problèmes :

– le programmeur risque d'oublier d'entourer ses instructions d'affichage avec les expressions ci-dessus, en particulier d'oublier le *unwind-protect*,

– le lecteur du code doit reconnaître ces expressions comme faisant partie d'un idiome,
– cela induit quatre lignes de code supplémentaires à chaque fois.

C'est une situation similaire à celle qui provoque la décision d'écriture d'une fonction : une suite d'expressions est souvent répétée et pour éviter la répétition physique, on crée une fonction avec un nom qui suggère ce que fait la suite d'expressions. Mais ici, il est impossible d'en faire une fonction. La raison principale est que les *instructions à effectuer ne sont pas toujours les mêmes*. Ces instructions peuvent même contenir une boucle ou n'importe quel autre calcul arbitrairement compliqué.

Une macro, par contre est capable de capturer cet idiome. Par exemple, une telle macro permettra d'écrire :

```
(with-double-buffering screen
  calcul-1
  ...
  calcul-n)
```

Ici, l'idiome a été *nommé*, permettant facilement de ne pas l'oublier et de le reconnaître immédiatement, de la même façon qu'une fonction permet de reconnaître une suite d'instructions à exécuter.

Voici la macro Common Lisp permettant l'utilisation ci-dessus :

```
(defmacro with-double-buffering (screen &body body)
  `(progn
    (setf (pixmap ,screen) (allocate-pixmap))
    (unwind-protect
        (progn
          ,@body
          (copy-from-pixmap ,screen))
      (setf (pixmap ,screen) nil))))
```

Le reste de ce chapitre sera consacré à l'explication des divers composants de cette définition.

5.2 Macros basiques

Comme exemple simpliste de macro, supposons que nous souhaitions remplacer l'instruction

```
(setf *fuels*
      (list
       (list 'diesel 500)
       (list 'leaded 100)
       (list 'unleaded 200)))
```

par l'appel

```
(initialize-fuels *fuels*)
```

On suppose qu'il s'agit d'une application de gestion de stock de carburant et que `*fuels*` est une variable pouvant contenir une liste des carburants possibles et leur quantité disponible en hecto-litres.

Une macro Common Lisp permettant d'écrire `(initialize-fuels *fuels*)` aura pour nom *initialize-fuels* et acceptera un seul paramètre. Ce paramètre qui doit dans ce cas être un *symbole* n'est pas évalué (contrairement à ce qui se passe lors d'un appel de fonction). La définition de la macro aura la structure suivante :

```
(defmacro initialize-fuels (fuels)
  corps de la macro)
```

Dans le corps de la macro, le paramètre *fuels* aura comme valeur, le symbole **fuels** et la valeur renvoyée par l'appel à la macro, doit être la *liste* :

```
(setf *fuels*
      (list
        (list 'diesel 500)
        (list 'leaded 100)
        (list 'unleaded 200)))
```

Voici une façon d'y arriver :

```
(defmacro initialize-fuels (fuels)
  (list 'setf
        fuels
        '(list
          (list 'diesel 500)
          (list 'leaded 100)
          (list 'unleaded 200))))
```

On peut d'abord se convaincre que cette macro aura le bon effet :

```
* (defparameter *fuels1* nil)

*FUELS1*
* (defparameter *fuels2* nil)

*FUELS2*
* (setf *fuels1*
      (list
        (list 'diesel 500)
        (list 'leaded 100)
        (list 'unleaded 200)))

((DIESEL 500) (LEADED 100) (UNLEADED 200))
* (initialize-fuels *fuels2*)

((DIESEL 500) (LEADED 100) (UNLEADED 200))
* (equal *fuels1* *fuels2*)

T
*
```

On peut aussi utiliser la fonction *macroexpand-1* pour visualiser l'expansion de l'appel à la macro :

```
* (macroexpand-1 '(initialize-fuels *fuels2*))

(SETF *FUELS2*
        (LIST (LIST 'DIESEL 500)
```



```
              (LIST 'LEADED 100)
              (LIST 'UNLEADED 200)
T
*
```

Cette méthode de création de l'expression finale est possible uniquement pour de très petites expressions. La difficulté vient du mélange entre les parties variables (le paramètre *fuels*) et les parties constantes (le reste).

Pour faciliter la manipulation d'expressions, il existe une fonctionnalité en Common Lisp appelée *backquote*. Avec cette fonctionnalité, il est possible de créer l'expression finale à partir d'un *patron*. Son utilisation est similaire à celle de *quote*. Pour *backquote* il y a un caractère, l'apostrophe à l'envers, à utiliser pour signifier le début du patron. À l'intérieur d'un patron, toute expression précédée par le caractère *virgule* sera *évaluée* et remplacée par sa valeur. L'exemple ci-dessus devient alors :

```
(defmacro initialize-fuels (fuels)
  `(setf ,fuels
    (list
     (list 'diesel 500)
     (list 'leaded 100)
     (list 'unleaded 200))))
```

Maintenant, le corps de la macro ressemble à l'expression à renvoyer et ce sont uniquement les parties précédées du caractère *virgule* qui seront remplacées.

En réalité, le caractère *backquote* (apostrophe à l'envers) est reconnu par *read* qui est chargé de créer une expression similaire à celle utilisée dans la première version de la macro.

Il est parfois souhaitable de substituer une *suite* d'expressions dans un patron. C'est le cas lorsque la macro accepte un nombre arbitraire d'arguments. Dans ce cas, il convient d'utiliser le symbole *&body* dans la lambda-liste de la macro. Bien que synonyme de *&rest*, le symbole *&body* est préférable pour l'écriture de macros, car on parle souvent du corps (en anglais : body) de la macro. Pour substituer une suite d'expressions dans le patron, il n'est pas possible d'utiliser *virgule*, car on obtient alors une *liste* des expressions plutôt que la suite elle-même. Pour cette raison, la fonctionnalité *backquote* autorise aussi la suite des deux caractères `,@`.

Voici un exemple d'utilisation de *&body* et `,@`. Il s'agit d'écrire une macro permettant une boucle *repeat-until* similaire à celle du langage Pascal ou à la boucle *do-while* du langage C. Cette macro accepte en paramètre la condition d'arrêt de la boucle, la valeur de retour suivies d'un nombre arbitraire d'expressions correspondant au corps de la boucle.

```
(defmacro repeat-until (condition return-exp &body body)
  `(progn
    ,@body
    (do ()
        (,condition ,return-exp)
      ,@body)))
```

Voilà comment s'en servir :

```
* (defparameter *i* 0)

*I*
* (repeat-until (> *i* 4) 'ok
    (print *i*)
    (incf *i*))

0
1
2
3
4
OK
*
```

On peut vérifier l'expansion de cet appel :

```
* (macroexpand-1
    '(repeat-until (> *i* 4) 'ok
      (print *i*)
```



```
      (incf *i*)))

(PROGN (PRINT *I*) (INCF *I*)
       (DO () ((> *I* 4) 'OK) (PRINT *I*) (INCF *I*)))
T
*
```

Indentée différemment, cette expression peut s'écrire :

```
(PROGN
  (PRINT *I*)
  (INCF *I*)
  (DO ()
      ((> *I* 4) 'OK)
    (PRINT *I*)
    (INCF *I*)))
```

On voit alors que les `,@body` ont été remplacés par *une suite de deux expressions*, ce qui n'aurait pas été possible avec *virgule* seulement.

Nous verrons section 6.2.2 que cette solution n'est en fait pas bonne du fait que *,@body* est présent deux fois dans le corps de la macro.

5.3 Exemples

5.3.1 Un nouveau type de boucle

Supposons que le programmeur d'une application particulière soit très attaché à la boucle *while* du langage Pascal. Il a donc envie d'écrire des expressions du type :

```
(setf i 0)
(while (< i n)
  (setf (aref tab i) 0)
  (incf i))
```

Voici une macro permettant de le faire extraite de [3] :

```
(defmacro while (test &body body)
  `(do ()
       ((not ,test))
     ,@body))
```

Voici comment s'en servir :

```
* (setf *i* 0)

0
* (while (< *i* 4)
    (print *i*)
    (incf *i*))

0
1
2
3
NIL
*
```

Attention, si on utilise Common Lisp sous l'éditeur Emacs, l'indentation par défaut proposée par celui-ci n'est pas celle indiquée dans l'exemple ci-dessus, car Emacs ne connaît pas l'existence de la macro *while*. Il conviendra donc d'*apprendre* à Emacs cette nouvelle indentation. Pour la macro *while*, il y a une expression à indenter spécifiquement. Il faudra donc mettre dans le fichier `.emacs` l'expression Emacs Lisp suivante :

```
(put 'while 'lisp-indent-function 1)
```

De manière générale, dans le cadre de la création d'une application, il sera souvent nécessaire d'apprendre à Emacs l'indentation des macros correspondant aux idiomes de l'application.

5.3.2 Création de contexte

Parfois, on souhaite évaluer une suite d'expressions dans un contexte temporaire dont la création dépend de l'application. Par exemple, dans une ap-

plication de manipulation de formes géométriques, on souhaite parfois créer un contexte contenant les coordonnées polaires d'un point. Au lieu d'écrire :

```
(let ((rho (module point))
      (theta (angle point)))
  expression-1
  ...
  expression-n)
```

on souhaite pouvoir écrire :

```
(with-polar-coordinates (rho theta) point
  expression-1
  ...
  expression-n)
```

On peut envisager la macro suivante :

```
(defmacro with-polar-coordinates ((rho theta) point &body body)
  `(let ((,rho (module ,point))
         (,theta (angle ,point)))
    ,@body))
```

On remarque ici une fonctionnalité supplémentaire concernant la lambda-liste d'une macro. À la place d'un paramètre ordinaire, on trouve une liste de paramètres. Ceci permet une *destructuration* automatique de l'argument de la macro.

```
* (macroexpand-1 '(with-polar-coordinates (m a) *p* e1 e2 e3))

(LET ((M (MODULE *P*))) (A (ANGLE *p*))
  E1
  E2
  E3)
T
*
```

Ici encore nous devrons apprendre à Emacs l'indentation de cette nouvelle construction en rajoutant dans le fichier `.emacs` l'expression Emacs Lisp :

```
(put with-polar-coordinates 'lisp-indent-function 2)
```

Chapitre 6

Notions avancées

Rares étant les langages disposant d'un système de macros, peu de programmeurs sont aguerris à l'écriture de telles abstractions. Or, ce type de programmation présente des difficultés tout à fait spécifiques de même qu'il demande un style de programmation adapté. Nous développons cela dans les sections suivantes.

6.1 Difficultés d'écriture

Deux problèmes majeurs se posent lors de l'écriture de macros. Il sont connus sous le nom de *capture de variable* et *évaluation multiple*.

6.1.1 Capture de variable

Supposons que nous souhaitions créer une macro *dochar* qui évalue une suite d'expressions pour chaque élément d'une chaîne de caractères. Une telle macro peut être utilisée de la manière suivante :

```
* (dochar char "abc"
    (print char))

#\a
#\b
```



```
#\c
NIL
*
```

Voici une tentative d'écriture d'une telle macro :

```
(defmacro dochar (char string &body body)
  `(do ((i 0 (1+ i)))
        ((>= i (length ,string)))
     (let ((,char (aref ,string i)))
       ,@body)))
```

La documentation de cette macro pourrait être : « cette macro accepte une variable, une expression dont la valeur doit être une chaîne de caractères et une suite d'expressions qui seront évaluées avec la variable ayant comme valeur chaque élément de la liste. »

Mais il y a un problème : la macro ne s'exécute pas toujours conformément à la documentation. Voici d'abord un exemple qui marche :

```
* (let ((x 32))
    (print x)
    (dochar char "abc"
        (print (code-char (- (char-code char) x)))))

32
#\A
#\B
#\C
NIL
*
```

Voici maintenant un exemple similaire, mais qui ne marche pas :

```
* (let ((i 32))
    (print i)
    (dochar char "abc"
        (print (code-char (- (char-code char) i)))))
```

```
32
#\a
#\a
#\a
NIL
*
```

L'utilisateur de la macro s'attend probablement à ce que ces deux exemples donnent le même résultat. Après tout, la seule différence est le nom de la variable locale : `x` dans le premier exemple et `i` dans le deuxième. Pourtant les deux résultats sont complètement différents. Étant donnée la documentation, l'utilisateur s'attend probablement au résultat du premier exemple et pas à celui du deuxième.

Le second exemple échoue du fait qu'il existe *deux* variables différentes dont le nom est *i*. Le second exemple est équivalent à :

```
(let ((i 32))
  (print i)
  (do ((i 0 (1+ i)))
      ((>= i (length "abc")))
    (let ((char (aref "abc" i)))
      (print (code-char (- (char-code char) i))))))
```

Mais la boucle *do* introduit une deuxième variable *i*. Si l'on met cette deuxième variable en majuscule, on obtient :

```
(let ((i 32))
  (print i)
  (do ((I 0 (1+ I)))
      ((>= I (length "abc")))
    (let ((char (aref "abc" I)))
      (print (code-char (- (char-code char) I))))))
```

Le problème est que dans l'expression

```
(print (code-char (- (char-code char) i)))
```


c'est le *I* majuscule qui est utilisé et non (comme le pensait l'utilisateur) le *i* minuscule. Ce phénomène est appelé *capture de variable* et c'est le plus souvent (mais pas toujours) un phénomène indésirable.

Il est bien sûr possible de modifier la documentation pour qu'elle soit conforme au comportement de la macro en rajoutant : « Attention : ne pas utiliser la variable *i* dans les expressions de la suite ». Mais c'est une solution peu satisfaisante.

La bonne solution est d'utiliser la possibilité Common Lisp de *créer des symboles* garantis différents de tout autre symbole utilisé dans le programme. La fonction *gensym* renvoie à chaque appel un nouveau symbole qui ne peut pas être confondu avec un symbole explicitement tapé par le programmeur. Voici comment modifier l'exemple pour éviter la capture :

```
(defmacro dochar (char string &body body)
  (let ((var (gensym)))
    `(do ((,var 0 (1+ ,var)))
         ((>= ,var (length ,string)))
       (let ((,char (aref ,string ,var)))
         ,@body))))
```

Cet exemple illustre aussi la possibilité d'effectuer un calcul préalable à la création de l'expansion. Ici, la partie :

```
(let ((var (gensym)))
```

est exécutée par le compilateur pendant la macro-expansion. L'effet de cette expression est de créer un nouveau symbole. Ce symbole sera la valeur de la variable locale *varl* qui n'existe *que* pendant la macro-expansion. Dans le patron, la variable *var* est précédée d'une virgule, ce qui signifie qu'elle sera évaluée. Chaque occurrence de *var* dans le patron sera donc remplacée par le symbole généré par *gensym*. Ce symbole servira donc de variable dans l'expansion finale. Il est important de réaliser qu'une nouvelle variable est générée à chaque fois que la macro est appelée. De cette façon, il est possible d'imbriquer des appels à la macro sans risque de capture.

Avec cette nouvelle définition, le deuxième exemple marche aussi.

6.1.2 Évaluation multiple

Prenons l'exemple de macro suivant :

```
(defmacro set-var (x val)
  `(progn
    (setf ,x ,val)
    (format t "Side effect ~A := ~A~%" ',x ,val)))
```

Cet exemple pose un grave problème qui apparaît lorsque l'expression fournie pour la valeur `val` n'est pas une simple variable, mais une expression avec des effets de bord. Voici un exemple :

```
* (defparameter *val* -1)

*val*
* (defparameter *tab* (make-array 3))

*tab*
* (set-var (aref *tab* 0) (incf *val*))

Side effect (AREF *TAB* 0) := 1
NIL
* (set-var (aref *tab* 1) (incf *val*))
Side effect (AREF *TAB* 1) := 3
NIL
* (set-var (aref *tab* 2) (incf *val*))
Side effect (AREF *TAB* 0) := 5
NIL
* *tab*

#(0 2 4)
*
```

Ici, on s'attendrait à ce que les éléments du tableau prennent les valeurs respectives *0, 1, 2*. Pourtant, ils prennent les valeurs *0, 2, 4* tandis que les valeurs affichées sont *1, 3, 5*. Le problème est évident quand on regarde la macro-expansion :


```
* (macroexpand-1 '(set-var (aref *tab* 0) (incf *val*)))

(PROGN
 (SETF (AREF *TAB* 0) (INCF *VAL*))
 (FORMAT T "Side effect ~A := ~A~%" '(AREF *TAB* 0) (INCF *VAL*)))
T
*
```

On remarque que l'expression :

```
(incf *val*)
```

non seulement figure deux fois dans l'expansion, mais sera évaluée *deux fois*.

Ce phénomène s'appelle *évaluation multiple* et comme le phénomène de capture de variable, il est le plus souvent indésirable. L'utilisateur s'attend probablement à ce que l'expression concernant le vecteur soit évaluée une seule fois avant le début de l'exécution de la boucle.

Voici la solution du problème :

```
(defmacro set-var (x val)
  (let ((value (gensym)))
    `(let ((,value ,val))
      (setf ,x ,value)
      (format t "Side effect ~A := ~A~%" ',x ,value))))
```

Ici, une deuxième variable générée *value* a été introduite. Elle contiendra la valeur de l'expression *val*, qui ne sera donc évaluée qu'une seule fois. Dans le *let* cette nouvelle variable est utilisée à la place de l'expression initiale. Maintenant, l'exemple fonctionne correctement :

```
* (defparameter *val* -1)

*VAL*
* (set-var (aref *tab* 0) (incf *val*))
Side effect (AREF *TAB* 0) := 0
NIL
```

```
* (set-var (aref *tab* 1) (incf *val*))
Side effect (AREF *TAB* 1) := 1
NIL
* (set-var (aref *tab* 2) (incf *val*))
Side effect (AREF *TAB* 2) := 2
NIL
* *tab*

#(0 1 2)
*
```

Il est instructif de regarder l'expansion d'une telle expression :

```
* (macroexpand-1
   '(set-var (aref *tab* 0) (incf *val*)))

(LET ((#:G4792 (INCF *VAL*)))
  (SETF (AREF *TAB* 0) #:G4792)
  (FORMAT T "Side effect ~A := ~A~%" '(AREF *TAB* 0) #:G4792))
T
*
```

On voit alors immédiatement que l'expression `(incf *val*)` ne sera évaluée qu'une seule fois.

6.2 Style d'écriture

Pour écrire des macros le programmeur doit adopter un style différent de celui qu'il utilise pour écrire des fonctions.

6.2.1 Deux styles de programmation mélangés

Dans la partie de la macro exécutée pendant l'expansion, le style doit être dense, plutôt fonctionnel mais pas forcément efficace, tandis que dans la partie générée comme résultat de l'expansion, le style doit avoir la performance comme objectif principal.

6.2.2 Taille du code produit

Cependant le programmeur doit veiller à ne pas faire exploser la taille du code produit. Le risque majeur encouru est d'augmenter le taux de défauts de cache à l'exécution et donc de détériorer considérablement les temps d'exécution. Le temps de compilation (après macro-expansion) aussi peut devenir rédhibitoire.

Reprenons, l'exemple de la boucle *repeat-until* de la page 35. Le nombre d'expressions que l'utilisateur peut fournir par l'intermédiaire de l'argument *body* n'est pas limité. Dans la solution de la page 5.2, ces expressions étant recopiées deux fois lors de la macro-expansion la taille du code produit est au moins le double de la taille du code initial (avant macro-expansion).

Étant donné que les appels de macros peuvent être imbriqués à une profondeur arbitraire, si on laisse les macros faire de multiples copies de leur argument *body* (de taille non bornée), on prend le risque d'obtenir du code de taille exponentielle ; ceci est à éviter pour les raisons de performance citées ci-dessus.

La duplication de l'argument *body* dans le corps d'une macro est donc à proscrire absolument. La solution consiste à définir dans le code produit une fonction auxiliaire sans argument (à l'aide d'un *flet*) dont le rôle est d'exécuter les expressions contenues dans *body* et d'y faire appel à chaque fois que nécessaire.

Pour illustrer cette technique nous donnons ici la version correcte de la boucle *repeat-until*.

```
(defmacro repeat-until (condition return-exp &body expressions)
  (let ((f (gensym)))
    `(flet ((,f () ,@expressions))
      (,f)
      (do ()
          (,condition ,return-exp)
        (,f)))))
```

6.2.3 Contrôle de l'évaluation

En Common Lisp, lors d'un appel de fonction Common Lisp, tous les arguments sont systématiquement évalués. En revanche, par défaut, les arguments d'une macros ne sont pas évalués. Ce fait peut être exploité pour contrôler plus finement l'évaluation des arguments d'un opérateur.

Par exemple, on souhaite écrire un prédicat `none` qui s'applique à un nombre arbitraire d'expressions $e_1, \ldots, e_n$, qui retourne vrai si toutes les expressions sont fausses et faux sinon. Dans le cas où une des expressions e_i renvoie vrai, on ne souhaite pas que les expressions $e_{i+1}, \ldots, e_n$ soient évaluées on écrit donc une macro plutôt qu'une fonction :

```
(defmacro none (&body body)
  `(not (or ,@body)))
```

Cette macro fait appel à la macro *or* qui n'évalue pas systématiquement tous ses arguments puisque l'évaluation s'arrête dès qu'un des arguments s'évalue en vrai.

6.2.4 Un dernier exemple

On souhaite définir une opération *def-sequencer* qui à partir d'un *nom* et d'une liste de valeurs *l* définit la fonction globale de nom *nom* laquelle renvoie à chaque appel la valeur suivante dans la liste et ceci de manière circulaire.

On souhaite par exemple pouvoir écrire l'appel

```
(def-sequencer seq '(do mi sol mi))
```

puis utiliser la fonction *seq* :

```
* (def-sequencer seq '(do mi sol mi))

SEQ
* (seq)

DO
```



```
* (seq)

MI
* (seq)

SOL
* (seq)
```

L'appel `(def-sequencer seq '(do mi sol mi))` doit avoir le même effet que si on avait effectué la définition suivante :

```
(let ((new (circ (copy-list '(do mi sol mi)))))
  (defun seq ()
    (prog1
        (car new)
      (setf new (cdr new)))))
```

en supposant qu'on ait au préalable défini une opération *circ* permettant de rendre une liste circulaire :

```
(defun circ (l)
  (nconc l l))
```

Une solution possible pour la macro *def-sequencer* est :

```
(defmacro def-sequencer (nom l)
  (let ((new (gensym)))
    `(let ((,new (circ (copy-list ,l))))
      (defun ,nom ()
        (prog1
            (car ,new)
          (setf ,new (cdr ,new)))))))
```

On peut ensuite définir une macro *with-sequencer* qui permet de définir et d'utiliser localement une fonction définie par *def-sequencer*. Voici un exemple d'utilisation :

```
(with-sequencer next '(sol la si do re mi fa#)
  (let ((l '()))
    (dotimes (i 10 (nreverse l))
      (push (next) l))))
```

et voici la macro *with-sequencer* :

```
(defmacro with-sequencer (name l &body body)
  `(progn
    (def-sequencer ,name ,l)
    ,@body))
```

Troisième partie

Exemples de langages enchâssés

Chapitre 7

Une version préfixe du langage PostScript

7.1 Introduction

Considérons une application ayant besoin de dessiner des figures construites d'une façon similaire à celle utilisée par PostScript [1]. PostScript maintient un environnement graphique contenant une transformation courante, un point de dessin courant et un chemin courant.

Le dessin se fait en deux phases. Un chemin est d' abord construit, puis tracé. Les commandes de construction de chemin sont `moveto`, `lineto`, `arcto` et `curveto`. La commande `moveto` crée un nouveau sous-chemin alors que les autres rajoutent un segment au sous-chemin courant.

Le chemin courant est exprimé avec les coordonnées du repère physique (de l'imprimante ou de l'écran), tandis que les commandes de construction de chemin sont exprimées avec les coordonnées du repère utilisateur courant. C'est la transformation courante qui est chargée de la traduction entre les deux.

Certaines commandes ont pour but de modifier la transformation courante. C'est le cas de `translate`, `scale` et `rotate`.

Le traçage du chemin courant se fait grâce à la commande `stroke`.

Notre petit langage sera appelé *prescript* car il utilise la notation préfixe contrairement à PostScript qui utilise la notation postfixe.

7.2 Définition du petit langage

Voici une spécification assez informelle du langage *prescript*. Notre implémentation ne contiendra pas l'ensemble des commandes de construction de chemin : seules les commandes `moveto` et `lineto` seront implémentées.

Le langage manipule une *pile d'environnements*. Chaque environnement contient un *chemin courant*, une *transformation courante* et un *point courant*.

7.2.1 Construction du chemin

Le chemin est initialement vide. En général, il contient un nombre arbitraire de sous-chemins. Chaque sous-chemin est tracé séparément. Un sous-chemin est une liste d'éléments. Le premier élément du sous chemin est un point indiquant le début du chemin. Les autres éléments sont des segments de droites, d'arcs ou de courbes de Bézier (seuls les segments de droites seront implémentés). Un segment commence implicitement au point de terminaison du segment précédent ou au point initial s'il s'agit du premier segment.

La commande `moveto` débute un nouveau sous-chemin lequel sera initialisé avec le point donné en argument de la commande.

La commande `lineto` rajoute un segment de droite à la fin du sous-chemin courant. L'argument de cette commande est le point extrémité du segment, le point origine étant le point courant avant l'exécution de la commande.

7.2.2 Traçage du chemin

Le chemin courant est tracé grâce à la commande `stroke`. Cette commande n'a aucun argument.

7.2.3 La transformation courante

Le chemin courant contient des coordonnées physiques. Les commandes de construction du chemin prennent comme arguments des coordonnées dans un repère différent et modifiable par l'utilisateur. C'est la transformation courante qui est chargée de la traduction entre les deux repères.

La transformation courante est une transformation *affine* permettant une combinaison arbitraire de rotations, translations et modifications d'échelle. Elle est exprimée par une matrice agissant sur les coordonnées acceptés par les commandes de construction de chemin.

La transformation courante peut être modifiée grâce aux commandes `rotate` (rotation du repère courant d'un angle donné), `scale` (transformation de l'échelle du repère courant de facteurs donnés) et `translate` (translation du repère courant d'un vecteur donné).

Une modification de la transformation courante influence les commandes de construction de chemin qui sont exécutées *après* la modification uniquement.

7.2.4 Modification de la pile d'environnements

La pile d'environnements peut être manipulée grâce à la commande `gsave`. Cette commande accepte un nombre arbitraire d'autres commandes. Ces commandes seront exécutées avec une pile modifiée. En fait, `gsave` duplique le sommet de la pile, exécute les commandes, puis dépile le sommet. De cette manière, la transformation courante, le chemin courant et le point courant peuvent être localement modifiés sans effet global.

7.2.5 Exemples

Voici quelques exemples d'utilisation du petit langage *prescript*.

Le premier exemple n'utilise que des arguments constants. L'interprète développé plus tard sera capable de traiter ce type de programmes :

```
...
(translate 100 20)
(moveto 0 0)
```



```
(gsave (lineto 0 200) (stroke))
(lineto 100 100)
(stroke)
...
```

L'interprète aura déjà du mal à interpréter des programmes contenant des expressions arithmétiques comme celle-ci :

```
...
(translate 100 20)
(moveto 0 0)
(gsave (lineto 0 200) (stroke))
(lineto 100 (+ 50 50))
(stroke)
...
```

En utilisant `eval` dans l'interprète, cette situation simple peut être réglée, mais il sera quand-même incapable de traiter des programmes contenant des références à des variables locales comme dans ce programme :

```
(let ((x 10) (y 20))
  ...
  (translate 100 y)
  (moveto 0 0)
  (gsave (lineto 0 200) (stroke))
  (lineto 100 (* x 10))
  (stroke))
```

Dans le programme précédent, le petit langage permet non seulement l'intégration d'expression Common Lisp dans des arguments de fonctions, mais aussi l'intégration du petit langage dans des expressions arbitraires Common Lisp.

L'exemple final montre la possibilité de mélanger librement plusieurs petits langages, ici *prescript* et `loop` :

```
...
```

```
(translate 100 100)
(moveto 400 400)
(lineto 430 460)
(stroke))
(moveto 0 0)
  (loop for i from 0 to 100 by 10
        do (gsave (lineto i (- 200 i)) (stroke)))
...
```

7.3 Le support commun

Afin de pouvoir manipuler les objets nécessaires pour notre petit langage, nous avons besoin d'un support commun. Le voici :

7.3.1 Le support X11

Le code décrit ci-dessous permet l'ouverture d'une fenêtre X11 sur laquelle le résultat la commande `stroke` peut être affiché et donc vérifié.

Le code utilise la bibliothèque CLX, laquelle est une implémentation en Common Lisp du protocole X11. Elle se situe donc au même niveau d'abstraction que la Xlib pour le langage C.

Remarque : CLX n'est pas une interface à la Xlib. Le protocole X11 est déjà bien adapté au langage Common Lisp et alors qu'il est relativement mal adapté au langage C (arguments facultatifs, arguments nommés avec mots clés, etc). Par conséquent, CLX est entièrement écrit en Common Lisp et communique directement avec le protocole X11.

Le code donné ici n'est pas complet dans le sens qu'il ne permet pas la fermeture de la fenêtre, ni la vérification d'erreurs. Il sert juste de support au reste de notre développement.

```
(defparameter *dpy* nil)
(defparameter *win* nil)
(defparameter *gctxt* nil)

(defun get-environment-variable (string)
```



```
  (cdr (assoc string ext:*environment-list* :test #'string=)))

(defun parse-display-variable (s)
  (let* ((colon (position #\: s))
         (dot (position #\. s :start colon))
         (host-name (if (zerop colon) "localhost" (subseq s 0 colon)))
         (display-number (parse-integer s :start (1+ colon) :end dot)))
    (values host-name display-number)))

(defun open-window ()
  (multiple-value-bind (host display)
      (parse-display-variable (get-environment-variable "DISPLAY"))
    (let* ((dpy (xlib:open-display host :display display))
           (screen (xlib:display-default-screen dpy))
           (black (xlib:screen-black-pixel screen))
           (white (xlib:screen-white-pixel screen))
           (win (xlib:create-window :parent (xlib:screen-root screen)
                                    :background white
                                    :x 0 :y 0 :width 500 :height 500))
           (gcontext (xlib:create-gcontext :drawable win
                                           :background white
                                           :foreground black)))
      (xlib:map-window win)
      (xlib:display-force-output dpy)
      (setf *dpy* dpy
            *win* win
            *gctxt* gcontext))))

(defun draw-line (x1 y1 x2 y2)
  (xlib:draw-line *win* *gctxt* (round x1) (round y1) (round x2) (round y2))
  (xlib:display-force-output *dpy*))
```

7.3.2 Point

Le petit langage (*prescript*) a besoin de manipuler des points en deux dimensions. Nous avons choisi d'implémenter cette abstraction sous la forme d'une *classe*. Ici, c'est un peu exagéré, car une abstraction plus légère (une *structure*) pourrait être suffisante. La raison principale d'utiliser une classe est d'illustrer la programmation par objets en Common Lisp.

Les objets de type `point` sont immuables. Cela évite la nécessité de copier le point courant lors de l'exécution d'une commande `gsave`. Une modification du point courant passe alors forcément par le remplacement du point courant dans le sommet de la pile, par un nouveau point obtenu à partir de l'ancien point. On appelle ce style de programmation *programmation fonctionnelle*, car les sous-programmes se comportent presque comme des fonctions mathématiques (sans effets de bord).

```
(defclass point ()
  ((x :initarg :x)
   (y :initarg :y)))

(defun make-point (x y)
  (make-instance 'point :x x :y y))
```

On remarque la convention de créer un *constructeur* qui appelle `make-instance`. Le client doit éviter l'utilisation directe de *make-instance*. La raison est que *make-instance* rend visible un certain nombre de détails d'implémentation de la classe.

7.3.3 Transformation

Le code présenté dans cette section permet de créer des transformations affines, de composer deux transformations et de transformer un point un autre point à l'aide une transformation.

On remarque l'utilisation de la construction '#.' permettant l'évaluation d'une expression pendant la phase de lecture par `read`, ainsi que l'utilisation de valeurs multiples pour récupérer les composantes d'une transformation.

Les objets de type `transformation` sont immuables. Comme dans le cas du point courant, cela évite la nécessité de copier la transformation courante lors de l'exécution de la commande `gsave`.

```
(defclass transformation ()
  ((mxx :initarg :mxx)
   (mxy :initarg :mxy)
   (myx :initarg :myx)
```



```
   (myy :initarg :myy)
   (tx :initarg :tx)
   (ty :initarg :ty)))

(defun get-transformation (transformation)
  (with-slots (mxx mxy myx myy tx ty) transformation
    (values mxx mxy myx myy tx ty)))

(defun make-transformation (mxx mxy myx myy tx ty)
  (make-instance 'transformation
    :mxx mxx :mxy mxy :myx myx :myy myy :tx tx :ty ty))

(defun make-translation-transformation (tx ty)
  (make-transformation 1 0 0 1 tx ty))

(defun make-rotation-transformation (angle)
  (let ((radians (* angle #.(/ pi 180))))
    (let ((s (sin radians))
          (c (cos radians)))
      (make-transformation c (- s) s c 0 0))))

(defun make-scaling-transformation (sx sy)
  (make-transformation sx 0 0 sy 0 0))

(defun compose-transformations (t1 t2)
  (multiple-value-bind (mxx1 mxy1 myx1 myy1 tx1 ty1) (get-transformation t1)
    (multiple-value-bind (mxx2 mxy2 myx2 myy2 tx2 ty2) (get-transformation t2)
      (make-transformation (+ (* mxx1 mxx2) (* mxy1 myx2))
                           (+ (* mxx1 mxy2) (* mxy1 myy2))
                           (+ (* myx1 mxx2) (* myy1 myx2))
                           (+ (* myx1 mxy2) (* myy1 myy2))
                           (+ (* mxx1 tx2) (* mxy1 ty2) tx1)
                           (+ (* myx1 tx2) (* myy1 ty2) ty1)))))

(defun transform-point (point transformation)
  (with-slots (mxx mxy myx myy tx ty) transformation
    (with-slots (x y) point
      (make-point (+ (* mxx x) (* mxy y) tx)
                  (+ (* myx x) (* myy y) ty)))))
```

7.3.4 Chemin

Comme le point courant et la transformation courante, le chemin courant doit être immuable si on veut que la commande `gsave` soit rapide.

Pour l'implémentation du chemin, nous utilisons le fait que les opérations Common Lisp de base sur les listes sont sans effets de bord. Un chemin est alors implémenté sous forme de liste de sous-chemins. Chaque sous-chemin est implémenté sous forme de liste d'éléments. Le premier élément de chaque sous-chemin est un point (le premier point du chemin) tandis que les autres sont des segments (de droites, d'arcs ou de courbes).

Afin de rendre les opérations rapides, nous avons choisi de stocker les sous-chemins d'un chemin dans l'ordre inverse. Le premier sous-chemin est alors le dernier élément de la liste. De même les éléments d'un sous-chemin sont stockés en sens inverse. Le point initial est donc le dernier élément de la liste et le dernier élément rajouté est le premier élément de la liste. Cela nous permet d'utiliser `cons` pour la construction de chemins et de sous-chemins. Cette fonction étant sans effets de bord, nous n'avons bas besoin de copier le chemin lors de l'exécution de la commande `gsave`.

On remarque que pour tracer un chemin, on le renverse d'abord, ce qui n'est pas strictement nécessaire. Chaque sous-chemin doit par contre être renversé.

Nous avons utilisé la fonction Common Lisp `reduce` pour parcourir les éléments d'un sous-chemin. Cette fonction permet de travailler sur chaque élément d'une liste avec en accumulant une valeur (dans notre cas, le point courant).

On remarque aussi l'utilisation d'une fonction générique `stroke-one`. Cette fonction prend un point et un segment, trace le segment et renvoie comme valeur le nouveau point courant. Cette fonction sera spécialisée par des méthodes spécifiques pour chaque type de segment. Dans notre cas, seul le segment de droite sera implémenté.

```
;;; A path is a list of subpaths.  Each valid subpath contains
;;; a list of elements (line- , curve-, and arc segments) with the
;;; last element being a point, the origin of the subpath.

;;; create an uninitialized path
```



```
(defun new-path ()
  '())

;;; initialize a path with a point
(defun initialize-path (path point)
  (cons (list point) path))

;;; add an element to an initialized path
(defun add-element (path element)
  (assert (car path))
  (cons (cons element (car path)) (cdr path)))

(defgeneric stroke-one (point segment))

(defun stroke-subpath (subpath)
  (reduce #'stroke-one (cdr subpath) :initial-value (car subpath)))

(defun stroke-path (path)
  (mapc #'stroke-subpath (reverse (mapcar #'reverse path))))
```

7.3.5 Segments

Bien que dans notre cas seul le segment de droite est implémenté, il faut imaginer une implémentation de chaque type de segment (arc, courbe).

On remarque surtout la méthode implémentant la fonction générique `stroke-one`. Cette méthode est appelée lorsque le deuxième argument de `stroke-one` est de type `lineseg`.

```
(defclass lineseg ()
  ((point :initarg :point)))

(defun make-lineseg (point)
  (make-instance 'lineseg :point point))

(defmethod stroke-one (point (seg lineseg))
  (let ((p2 (slot-value seg 'point)))
    (draw-line (slot-value point 'x)
```

```
               (slot-value point 'y)
               (slot-value p2 'x)
               (slot-value p2 'y))
    p2))
```

7.3.6 Environnement

La pile des environnements contient des éléments de type environnement (`env`). Un environnement contient tout ce qu'il faut pour la construction et le traçage de chemins, à savoir le point courant, la transformation courante et le chemin courant.

Un objet de type `env` n'est pas immuable. Ses composants sont constamment modifiés par les commandes du langage *prescript*.

```
(defclass env ()
  ((position :initarg :position)
   (transformation :initarg :transformation)
   (path :initform (new-path) :initarg :path)))

(defun make-initial-env ()
  (make-instance 'env
    :position (make-point 0 0)
    :transformation (make-transformation 1 0 0 -1 0 500)))

(defun copy-env (env)
  (with-slots (position transformation path) env
    (make-instance 'env
      :position position :transformation transformation :path path)))
```

7.4 Un interprète pour *prescript*

Dans ce chapitre, nous allons développer un interprète pour le petit langage *prescript*.

Cet interprète nous permet d'écrire des programmes du type :


```
* (prescript-interpreter
    '((moveto 2 4)
      (lineto 20 10)
      (gsave (lineto 34 23) (stroke))
      (lineto 44 55)))

*
```

Seules des constantes peuvent être utilisées comme arguments des opérations *prescript*.

Voici le code de l'interprète :

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; The interpreter

(setf (get 'moveto 'prescript)
      #'(lambda (envstack x y)
          (with-slots (position transformation path) (car envstack)
            (let ((p (transform-point (make-point x y) transformation)))
              (setf position p)
              (setf path (cons (list p) path))))))

(setf (get 'scale 'prescript)
      #'(lambda (envstack sx sy)
          (with-slots (transformation) (car envstack)
            (setf transformation
                  (compose-transformations
                   transformation (make-scaling-transformation sx sy))))))

(setf (get 'translate 'prescript)
      #'(lambda (envstack tx ty)
          (with-slots (transformation) (car envstack)
            (setf transformation
                  (compose-transformations
                   transformation (make-translation-transformation tx ty))))))

(setf (get 'rotate 'prescript)
      #'(lambda (envstack angle)
```

```
          (with-slots (transformation) (car envstack)
            (setf transformation
                  (compose-transformations
                   transformation (make-rotation-transformation angle))))))

(setf (get 'newpath 'prescript)
      #'(lambda (envstack)
          (with-slots (path) (car envstack)
            (setf path (new-path)))))

(setf (get 'lineto 'prescript)
      #'(lambda (envstack x y)
          (with-slots (position transformation path) (car envstack)
            (let ((p (transform-point (make-point x y) transformation)))
              (setf path (add-element path (make-lineseg p)))
              (setf position p)))))

(setf (get 'stroke 'prescript)
      #'(lambda (envstack)
          (with-slots (path) (car envstack)
            (stroke-path path))))

(setf (get 'gsave 'prescript)
      #'(lambda (envstack &rest commands)
          (let ((newenvstack (cons (copy-env (car envstack)) envstack)))
            (loop for command in commands
                  do (apply (get (car command) 'prescript)
                            newenvstack
                            (cdr command))))))

(defun prescript-interpreter (commands)
  (let ((envstack (list (make-initial-env))))
    (loop for command in commands
          do (apply (get (car command) 'prescript)
                    envstack
                    (cdr command)))))
```

On remarque l'utilisation de la *programmation dirigée par les données* (en

Anglais : data-driven programming).

La version ci-dessus contient beaucoup de duplication de code. En particulier, l'idiome :

```
(setf (get 'machin 'prescript)
      #'(lambda (...)
          ...))
```

est répété pour chaque commande. Grâce aux macros Common Lisp, il est possible de capturer cet idiome sous la forme d'une construction syntaxique. Voici le code modifié :

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; Additional support

(defmacro defprescript (name arglist &body body)
  `(setf (get ',name 'prescript)
    #'(lambda (envstack ,@arglist)
        (with-slots (position transformation path) (car envstack)
          ,@body))))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; The interpreter

(defprescript moveto (x y)
  (let ((p (transform-point (make-point x y) transformation)))
    (setf position p)
    (setf path (cons (list p) path))))

(defprescript scale (sx sy)
  (setf transformation
        (compose-transformations
         transformation (make-scaling-transformation sx sy))))

(defprescript translate (tx ty)
  (setf transformation
        (compose-transformations
         transformation (make-translation-transformation tx ty))))
```

```
(defprescript rotate (angle)
  (setf transformation
        (compose-transformations
         transformation (make-rotation-transformation angle))))

(defprescript newpath ()
  (setf path (new-path)))

(defprescript lineto (x y)
  (let ((p (transform-point (make-point x y) transformation)))
    (setf path (add-element path (make-lineseg p)))
    (setf position p)))

(defprescript stroke ()
  (stroke-path path))

(defprescript gsave (&rest commands)
  (let ((newenvstack (cons (copy-env (car envstack)) envstack)))
    (loop for command in commands
          do (apply (get (car command) 'prescript)
                    newenvstack
                    (cdr command)))))

(defun prescript-interpreter (commands)
  (let ((envstack (list (make-initial-env))))
    (loop for command in commands
          do (apply (get (car command) 'prescript)
                    envstack
                    (cdr command)))))
```

7.5 Un langage enchâssé pour *prescript*

Dans ce chapitre, nous allons développer un compilateur pour *prescript*. Le code généré sera du code Common Lisp qui sera ensuite compilé par le compilateur Common Lisp.

Voici un exemple qui montre comment se servir de cette version :

```
* (prescript
   (moveto 3 4)
   (gsave
    (scale 2 3)
    (loop for i from 5 downto 0
          for j from 0 to 5
          do (lineto i j))
    (stroke))
   (lineto 1 1)
   (stroke))

*
```

Comme le montre l'existence d'une boucle `loop` à l'intérieur du bloc `prescript`, le nouveau langage est complètement intégré dans Common Lisp.

Ici, `prescript` est clairement une macro, car elle doit créer un contexte contenant la transformation courante et le point de dessin courant. Notre tâche est d'écrire la macro `prescript`.

À l'intérieur du bloc `prescript`, `moveto`, `scale`, `newpath` et `lineto` doivent de préférence être des fonctions pour que le programmeur puisse faire quelque chose comme ceci :

```
* (prescript
   (moveto 0 0)
   (mapc #'lineto '(1 2 3) '(5 4 3))
   (stroke))

*
```

Dans un premier temps, nous ne traitons pas le cas de `gsave`. La stratégie sera donc de créer, grâce à `labels`, des fonctions locales, disponibles à l'intérieur du bloc :

```
(defmacro prescript (&body body)
```

```
  `(labels
       ((moveto (x y)
          ...)
        (scale (sx sy)
          ...)
        ...
    ,@body
    (values)))
```

La construction `(values)` à la fin sert à marquer le fait que le bloc `prescript` ne génère pas de valeur utile.

Maintenant, il faut introduire la pile d'environnements.

```
(defmacro prescript (&body body)
  (let ((envstack (gensym)))
    `(let ((,envstack (list (make-initial-env))))
      (labels
          ((moveto (x y)
             ...)
           (scale (sx sy)
             ...)
           (translate (tx ty)
             ...)
           (rotate (angle)
             ...)
           (newpath ()
             ...)
           (lineto (x y)
             ...)
           (stroke ()
             ...)))
          ,@body
          (values)))))
```

On remarque l'utilisation de `gensym` pour éviter des problèmes de capture de variables.

Afin de simplifier le code des commandes *prescript*, on introduit deux macros utilitaires :


```
(defmacro with-current-env (envstack &body body)
  (let ((e (gensym)))
    `(let ((,e (car ,envstack)))
      (with-slots (position transformation path) ,e
        ,@body))))

(defmacro with-transformed-point ((x y p) &body body)
  `(let ((,p (transform-point (make-point ,x ,y) transformation)))
    ,@body))
```

La première permet d'exécuter une suite d'expressions dans un contexte où la variable `e` contient l'environnement courant, la variable `position` contient le point courant, la variable `transformation` contient la transformation courante et la variable `path` contient le chemin courant.

La deuxième permet d'exécuter une suite d'expressions dans un contexte dans lequel un point a été transformé par une transformation dont le nom est `transformation`. Elle est donc destinée à être utilisée dans le corps d'un appel à la macro `with-current-env`.

Ces deux macros capturent de manière intentionnelle des variables .

Avec ces macros supplémentaires, voici maintenant le code de la macro `prescript` :

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; The macro prescript

(defmacro prescript (&body body)
  (let ((envstack (gensym)))
    `(let ((,envstack (list (make-initial-env))))
      (labels
          ((moveto (x y)
             (with-current-env ,envstack
               (with-transformed-point (x y p)
                 (setf position p)
                 (setf path (initialize-path path p)))))
           (scale (sx sy)
             (with-current-env ,envstack
               (setf transformation
```

```
                  (compose-transformations
                   transformation (make-scaling-transformation sx sy)))))
          (translate (tx ty)
            (with-current-env ,envstack
              (setf transformation
                    (compose-transformations
                     transformation (make-translation-transformation tx ty)))))
          (rotate (angle)
            (with-current-env ,envstack
              (setf transformation
                    (compose-transformations
                     transformation (make-rotation-transformation angle)))))
          (newpath ()
            (with-current-env ,envstack
              (setf path (new-path))))
          (lineto (x y)
            (with-current-env ,envstack
              (with-transformed-point (x y p)
                (setf path (add-element path (make-lineseg p)))
                (setf position p))))
          (stroke ()
            (with-current-env ,envstack
              (stroke-path path))))
       ,@body
       (values)))))
```

Nous allons maintenant attaquer la partie la plus difficile, à savoir `gsave`. Il est impossible d'écrire `gsave` sous la forme d'une fonction. La raison est que l'environnement courant doit être modifié *avant* l'exécution de la première expression du corps de `gsave`. Par conséquent, il faut utiliser une macro.

La difficulté principale est que la macro `gsave` est imbriquée dans une autre macro, `prescript`. On peut dire que la macro `with-transformed-point` est imbriquée dans la macro `with-current-env` ci-dessus, mais c'est une imbrication d'appels uniquement. Dans le cas de `gsave`, la macro `prescript` doit *définir* la macro `gsave` et ceci d'une manière locale, analogue à `labels` pour les fonctions.

La solution est d'utiliser *macrolet*. Voici le code qui en résulte :


```
;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; The macro prescript

(defmacro prescript (&body body)
  (let ((envstack (gensym)))
    `(let ((,envstack (list (make-initial-env))))
      (labels
          ((moveto (x y)
             (with-current-env ,envstack
               (with-transformed-point (x y p)
                 (setf position p)
                 (setf path (initialize-path path p)))))
           (scale (sx sy)
             (with-current-env ,envstack
               (setf transformation
                     (compose-transformations
                      transformation (make-scaling-transformation sx sy)))))
           (translate (tx ty)
             (with-current-env ,envstack
               (setf transformation
                     (compose-transformations
                      transformation (make-translation-transformation tx ty)))
           (rotate (angle)
             (with-current-env ,envstack
               (setf transformation
                     (compose-transformations
                      transformation (make-rotation-transformation angle)))))
           (newpath ()
             (with-current-env ,envstack
               (setf path (new-path))))
           (lineto (x y)
             (with-current-env ,envstack
               (with-transformed-point (x y p)
                 (setf path (add-element path (make-lineseg p)))
                 (setf position p))))
           (stroke ()
             (with-current-env ,envstack
               (stroke-path path))))
        (macrolet ((gsave (&body body)
                     `(progn (push (copy-env (car ,',envstack))
```

```
                                   ,',envstack)
                       (unwind-protect (progn ,@body)
                         (pop ,',envstack)))))
          ,@body
          (values))))))
```

Maintenant, imaginons une évolution possible de la macro `prescript`. Un utilisateur sophistiqué souhaite peut-être pouvoir rajouter de nouvelles fonctions comme `curveto` ou `arcto`. Il convient alors de définir un langage enchâssé pour ce type d'extensions. Voici une possibilité :

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; Additional support

(defmacro with-transformed-point ((x y p) &body body)
  `(let ((,p (transform-point (make-point ,x ,y) transformation)))
    ,@body))

(defmacro with-current-env (envstack &body body)
  (let ((e (gensym)))
    `(let ((,e (car ,envstack)))
      (with-slots (position transformation path) ,e
        ,@body))))

(defmacro aif (test then &optional else)
  `(let ((it ,test))
    (if it ,then ,else)))

(defparameter *expanders* '())

(defmacro defprescript (name arglist &body body)
  `(aif (assoc ',name *expanders*)
    (setf (cdr it) '(,arglist ,@body))
    (push '(,name ,arglist ,@body) *expanders*)))

(defprescript moveto (x y)
  (with-transformed-point (x y p)
    (setf position p)
    (setf path (initialize-path path p))))
```



```
(defmacro modify-transformation (arg)
  `(setf transformation
         (compose-transformations
          transformation
          ,arg)))

(defprescript scale (sx sy)
  (modify-transformation (make-scaling-transformation sx sy)))

(defprescript translate (tx ty)
  (modify-transformation (make-translation-transformation tx ty)))

(defprescript rotate (angle)
  (modify-transformation (make-rotation-transformation angle)))

(defprescript newpath ()
  (setf path (new-path)))

(defprescript lineto (x y)
  (with-transformed-point (x y p)
    (setf path (add-element path (make-lineseg p)))
    (setf position p)))

(defprescript stroke ()
  (stroke-path path))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; The macro prescript

(defmacro prescript (&body body)
  (let ((envstack (gensym)))
    `(let ((,envstack (list (make-initial-env))))
      (labels ,(mapcar #'(lambda (expander)
                           (let ((name (car expander))
                                 (arglist (cadr expander))
                                 (body (cddr expander)))
                             `(,name ,arglist
                               (with-current-env ,envstack
                                 ,@body))))
```

```
                *expanders*)
        (macrolet ((gsave (&body body)
                    `(progn (push (copy-env (car ,',envstack))
                                  ,',envstack)
                      (unwind-protect (progn ,@body)
                        (pop ,',envstack)))))
          ,@body
          (values))))))
```

Bibliographie

[1] *PostScript Language tutorial and cookbook.* Adobe Systems, 1985.

[2] Robert Strandh et Irène Durand. *Traité de Programmation en Common Lisp.* Publication Méta Modulaire, 2001.

[3] Paul Graham. *On Lisp.* Prentice Hall, 1993.

[4] Paul Graham. *ANSI Common Lisp.* Prentice Hall, 1995.

[5] Sonya E. Keene. *Object-Oriented Programming in Common Lisp.* Addison Wesley, 1989.

[6] Peter Norvig. *Paradigms of Artificial Intelligence Programming : Case Studies in Common Lisp.* Morgan Kaufmann, 1991.

[7] Guy L. Steele. *Common LISP : The language.* Prentice Hall, 1989.

Index