Langages Enchâssés

Robert Strandh
professeur à l'Université Bordeaux 1

Irène Durand
maître de conférences à l'Université Bordeaux 1

Février 2003

9, Allée des Bois du Stade
33700 Mérignac, France
http ://www.metamodulaire.com

Couverture
Conception : Pierre Castéran
Réalisation : Robert Strandh

Imprimé par :
Aquitaine Copie Service et Distribution
571 Cours de la Libération
33400 Talence

Premier tirage, ? ? ? ?

Dépôt légal : ? ? ? ?

ISBN : ? ? ? ?

Table des matières

Chapitre 1

Introduction

1.1 Public

Ce livre est destiné aux étudiants du cycle Master d'informatique et aux élèves d'écoles ingénieurs en informatique qui s'orientent vers une formation de Génie Logiciel.

Il nécessite des connaissances préalables du langage Common Lisp ainsi que des notions de Génie Logiciel.

1.2 Objectifs

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

Le premier objectif du livre est donc de convaincre le lecteur de l'utilité de langages dédiés pour la constructions de gros logiciels de qualités.

Le deuxième objectif est de montrer qu'une façon très raisonnable de l'implémentation d'une tel langage est la construction d'un langage enchâssés.

Finalement, nous souhaitons montrer au lecteur comment se servir du système de macros Common Lisp pour l'implémentation de ce 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 et à été développé par John McCarthy et son groupe de recherche à MIT. bla bla bla...

Afin qu'un langage soit utilisable pour l'écriture de langages enchâssés, il doit avoir un système de macros très puissant. Les macros proposés par un mécanisme de substitution de texte comme le préprocesseur du langage C ou le macroprocesseur M4 ne sont pas assez puissants. M4 est un peu meilleur que le préprocesseur C, car il est capable d'évaluer des expressions arithmétiques. Le langage Common Lisp est actuellement le seul à proposer un système de macros pour lequel un langage général entier est disponible.

Rien n'empêche en principe à quelqu'un d'étendre M4 pour contenir 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 ce langage est que le format interne (sous la 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 sous la forme de texte, mais il traduit, de la même façon que le compilateur, ce texte en format interne. Le système des macros travail sur cette représentation interne. Il reçoit des expression sous cette forme en entrée, et il génère d'autres expressions sous cette forme en sortie.

Pour qu'un langage de programmation soit bien adapté à l'écriture de macros, il doit donc être capable de représenter ces propres programmes sous la forme de données. La famille Lisp des langages est unique dans ce sens. On peut donc dire que seule un langage qui ressemble assez étroitement à un langage de la famille Lisp est adapté, voire capable, d'être un langage hôte pour un système de macros puissant, et donc pour des langages enchâssés.

Aujourd'hui, le seul représentant raisonnable de la famille Lisp est le langage Common Lisp. Sa définition est le résultat de plusieurs décennies de réflexions et d'expérimentation. Il est donc juste de dire que nous n'avions pas de véritable choix de langage pour ce livre, car 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 <u>`soulignée`</u> sera utilisée pour les messages du système Common Lisp vers l'utilisateur.

Une police comme *ceci* 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 est concerné par les aspects du génie logiciel qui sont la raison de l'existence des langages enchâssés. Dans cette première partie, nous traitons aussi une partie de l'histoire de certains aspects du génie logiciel importants pour le reste du livre.

La deuxième partie du livre est entièrement consacré à l'écriture de macros Common Lisp. Les macros Common Lisp sont très puissantes (en fait, le langage Common Lisp entier est à la disposition du programmeur de macros. Il y a par conséquent des idiomes qui se sont développés dans la communauté et qui sont à respecter obligatoirement par le programmeur expert. Ces idiomes sont traités en détails dans cette partie.

La troisième partie contient des exemples de langages enchâssés. En particulier, nous parlons d'un langage similaire à PostScript pour la génération de pages de texte et de graphique imprimées. Nous suivons l'évolution de ce petit langage d'abord implémenté non 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

Il y a deux philosophie principales de la manière dont on doit créer un programme. La première façon, la programmation ascendante était très populaire dans les années 1970 parmi les adhérents du mouvement de la programmation structurée. La deuxième façon, la programmation descendante est une réaction de la première, et peut être considérée comme la base de la programmation par 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 fini. La manifestation principale de la programmation descendante est certainement la méthode proposée par le créateur du langage de programmation Pascal, Niclaus Wirth, qu'il a nommé 'raffinement par pas' (en anglais : stepwise refinement).

À chaque étape de la méthode, une description d'un niveau assez haut est raffiné en plusieurs descriptions d'un niveau un peu moins haut. Le processus se termine lorsque chaque description est d'un niveau assez bas pour correspondre à des instructions dans le langage de programmation utilisé. À ce stade, il suffit de traduire la spécification en code, et lorsque la méthode a

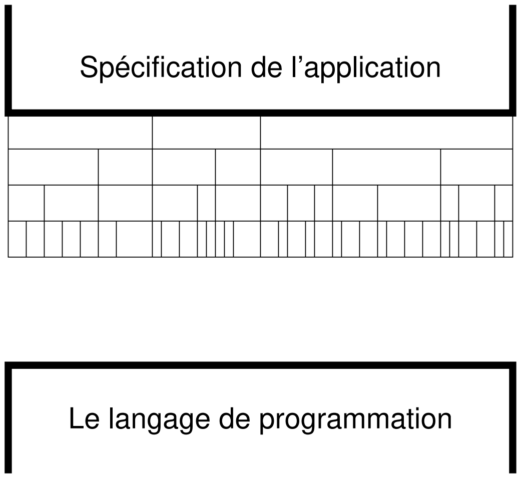

FIG. 2.1 – La programmation descendante

été appliquée à l'ensemble des sous descriptions, la spécification a été traduit en code, et le système est fini. Ce processus est illustré par la figure 2.1.

Les 'modules' quie résultent de la programmation ascendante correspondent souvent à une *fonctionnalité* du système. Il est raisonnable de demander de que *fait* un tel module.

La programmation descriptions a des inconvénients majeurs.

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. En fait, il est courant parmi les utilisateurs professionnels de cette méthode de diffuser la réalisation des méthodes de haut niveau sur différents membres de l'équipe de développement. Il semble donc inévitable qu'un module soit développé plusieurs fois, d'un façon légèrement différente à chaque fois, par plusieurs développeurs différents de modules différents.

Plus important, la programmation descendante implique la nécessité de prendre une décision détaillée concernant ce que va faire le système final (la spécification détaillée du système) très tôt dans le projet. Alors que cela peut paraître normal de savoir ce que doit faire le système avant de commencer le développement, ce fait est en réalité à l'origine d'un grand nombre d'échecs de grands systèmes informatiques.

Le problème principal est qu'il est quasi impossible d'exiger de la part du client une spécification détaillée du système final. Une spécification n'est d'abord jamais détaillé. On peut argumenter qu'une spécification détaillée est pratiquement équivalent a son code final, et donc impossible à réaliser par le client. Puis, c'est très difficile de savoir a priori ce dont on a besoin. La spécification évolue assez souvent pendant la durée de développement du système. Avec la programmation descendante, ce que doit faire le système est malheureusement décidé très tôt, et tout développement dépende de cette spécification initiale. Une fois le développement commencé, il est donc difficile, voire impossible de changer son idée de ce que doit faire le système sans invalider l'effort développement effectué jusqu'alors.

Nous pratiquons malheureusement toujours la programmation descendante, avec un résultat assez catastrophique. Selon des études de l'armée américaine, environ 9 systèmes sur 10 développés sur mesure ne sont jamais mise en service. Entre l'établissement de la spécification et la livraison du système, les souhaits du client ont tellement évolués que le système livré n'est plus d'actualité. Parmi les systèmes mises en services, une grande majorité l'est seulement après des révisions profondes selon les nouveaux souhaits du clients.

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 décider trop tôt les fonctionnalités du système.

Les modules de la programmation ascendante ne correspondent souvent pas à des fonctionnalités, mais à des *types de données* ou *types abstraits* dans le système final. On s'aperçoit que ces types sont plus stables que les fonctionnalités du système. Un système pour la 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.

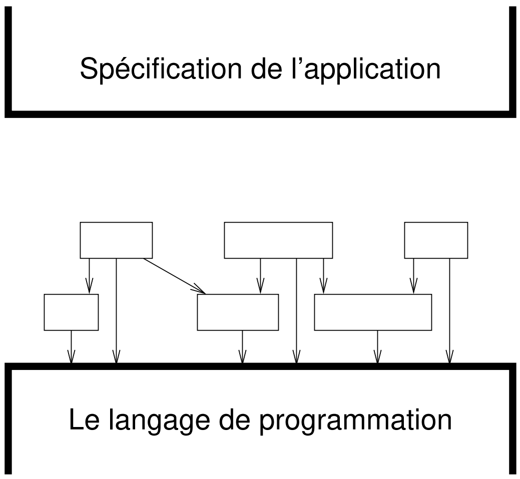

Fig. 2.2 – La programmation ascendante

Avec la programmation, le développeur construit les types abstrait d'abord à partir du langage de programmation 'nu', puis en réutilisant les 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 exprimés en utilisant le langage de programmation, et donc exécutables. Le degré de réutilisation des module 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 à évolué et s'utilise aujourd'hui sous la forme de la *programmation par objets*, qui rajoute 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 au langage de programmation pour l'écriture de systèmes de réservation de billets d'avions. Les modules ne sont pas spécialisés à un seul système, mais à une famille de systèmes (ou à un *domaine* d'applications) car les modules ont été réalisés sans tenir compte de ce que doit faire le système final en détail.

Il est évident que certains modules seront toujours spécialisés par rapport au système final, sinon, il ne peut difficilement réaliser la fonctionnalité souhaitée. Mais il est possible qu'une grande majorité des modules soit plus générale et que les modules 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 vu comme une extension à un langage de programmation, spécialisé pour un domaine d'applications.

Dans ce chapitre, nous prenons cette idée, et nous la poussons à l'extrême. L'idée est d'avoir un langage de programmation contenant déjà une grande partie des fonctionnalités nécessaires pour un domaine d'applications. Il est claire que s'il est possible pour une équipe de développement de se procurer d'un tel langage, son travail est en grande partie 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'autre cas, l'application sera un mélange entre le langage dédié et le langage hôte.

3.1 Qualités d'un langage dédié

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

On reconnaît que la programmation est une activité dont l'objectif est la construction systématique d'*abstractions*. Trois types d'abstractions sont reconnus aujourd'hui :

- l'abstraction fonctionnelle. Il s'agit de la construction de *procédures* et de *fonctions* pour capturer et nommer des idiomes pouvant s'exprimer de cette manière ;
- l'abstraction de données. C'est la construction de *types abstraits* dont l'application a besoin :
- l'abstraction syntaxique. Cette abstraction permet de nouvelles constructions syntaxiques, permettant ainsi 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 existants 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'applications. 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, il est impossible pour le compilateur de détecter des fautes qui sont faciles à détecter dans le cas d'une construction syntaxique.

Le langage graphique OpenGL est un bon exemple d'un 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 dont l'environnement graphique est manipulé grâce à des fonctions pour sauver et restaurer l'environnement actuel. Une erreur fréquente en PostScript est d'ailleurs de déborder la pile d'environnements, car il est facile d'oublier de restaurer l'environnement. Une construction syntaxique, sous la forme de parenthèses équilibrées permettrait au compilateur de détecter cette faute.

Mais l'exemple le plus flagrant est peut-être la construction d'interfaces graphiques qui se font presque exclusivement sous la forme d'une suite d'appels à des fonctions pour la 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. Voici quelques exemples :

- Le Passage de paramètres. Mise 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. Encore, la syntaxe spécifique permet au compilateur d'effectuer certaines vérifications qui ne seraient pas possible si la création de classes se faisait seulement avec des appels de fonctions. Mais la syntaxe permet surtout au programmeur d'exprimer de manière plus dense les constructions souhaitées.

Il y a là une lacune flagrante dans la plupart des langages de programmation existants. Ils reconnaissent l'importance de fonction et de procédures, de types de données et de constructions syntaxiques. Ils reconnaissent l'importance de donner au programmeur la possibilité de créer ses propres fonctions et procédures et ses propres types de données, mais ils ne considèrent pas important de permettre au programmeur de créer ses propres constructions syntaxiques. Cette situation est illustrée par la figure 3.1.

Dans la suite, nous allons voir que la possibilité pour le programmeur de créer ses propres constructions syntaxiques est cruciale pour l'implémentation des langages dédiés.

Le langage Common Lisp est parmi les rares langages permettant ce types de constructions. Il est donc juste de dire que ce langage est donc parmi les seuls à permettre la construction de langages enchâssés. La possibilité pour Common Lisp d'admettre les trois types d'abstractions est illustrée par la figure 3.2.

3.2 Techniques d'implémentation d'un langage dédié

Comment peut-on envisager l'implémentation d'un langage dédié ? Il y a plusieurs façons. Comme cela a été expliqué dans la section précédente, il

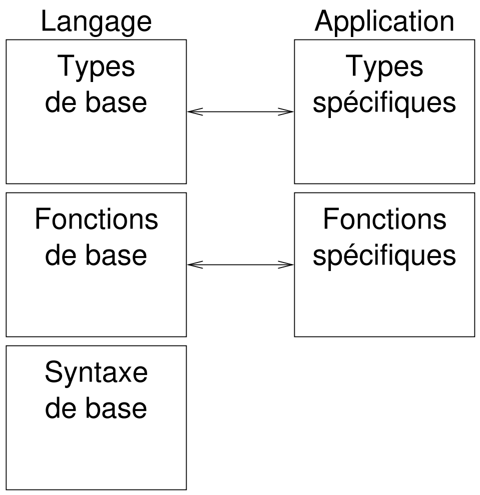

FIG. 3.1 – Abstractions dans les langages traditionnels

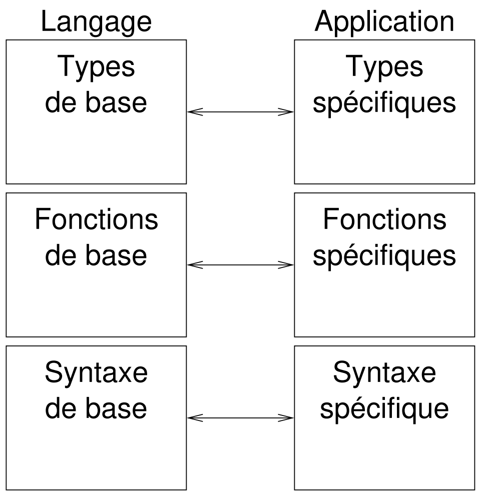

FIG. 3.2 – Abstractions dans Common Lisp

faut permettre des constructions syntaxiques spécifiques au domaine cible du langage. Une simple implémentation sous la forme de procédures, fonctions, classes, etc., d'un langage de programmation traditionnel ne le permet pas.

3.3 Préprocesseur

La première possibilité et de construire un *préprocesseur*. C'est la technique utilisé pour le langage Pro*C permettant 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 simples en langage C, souvent des appels de fonctions pour ouvrir, fermer et accéder à la base. Puis le programme qui en résulte est compilé par le compilateur C habituel.

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

Un avantage considérable de cette technique est la possibilité de réutiliser *en masse* les constructions de base du langage hôte. Ainsi, le créateur du préprocesseur n'a pas à être concerné par des fonctionnalités comme l'arithmétique, l'appel de fonctions, 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 pour le programmeur de Pro*C d'avoir 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 pour le préprocesseur d'avoir une connaissance si profonde du langage hôte.

Un autre problème existe avec cette technique lorsque l'on souhaite mélanger deux ou plusieurs langages dédiés. On est forcé de choisir un ordre entre l'application des préprocesseurs. Le problème est que le premier préprocesseur ne reconnaît pas forcement 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 le langage de l'autre.

3.4 Interprète

Une autre façon d'implémenter un langage dédié est d'écrire un interprète pour le langage. C'est le choix de certains langage 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.

Mais ici encore, il y a de nombreux inconvénients. D'abord il n'est pas possible de profiter des constructions de base du langage hôte, ici le langage 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 des construction arithmétiques, des boucles, etc. L'implémentation du langage dédié nécessite alors des connaissances assez générales concernant l'implémentation de langages de programmation.

De plus, il faut avoir le temps pour créer l'interprète. Il est considérablement plus difficile d'écrire un interprète que d'écrire un préprocesseur. Un investissement insuffisant dans le traitement d'erreurs peut facilement rendre l'interprète inutilisable pour d'autres personnes que ses créateurs.

Finalement, un interprète est plus lent en exécution que le code généré par un préprocesseur puis compilé par un compilateur natif pour le langage hôte. Beaucoup d'applications nécessite une exécution rapide du code. L'utilisation d'un langage dédié implémenté sous la forme d'un interprète peut nécessiter de nombreuses modifications de l'interprète pour augmenter la performance de certaines constructions, voir 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 sans possibilité d'avoir les deux simultanément.

3.5 Compilateur

La lenteur d'un interprète peut être éliminée si l'on implémente le langage dédié par une compilateur plutôt que par un interprète. Le compilateur peut générer du code en langage hôte afin d'éviter des dépendances du processeur.

Certains inconvénients avec l'interprète restent en place. La nécessité de dupliquer des constructions de base comme l'arithmétique ne changent pas avec cette méthode.

Par contre, 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, et il faut avoir une expérience avec diverses techniques de compilation.

3.6 Définition de langage enchâssé

Après la descriptions de plusieurs techniques d'implémentation de langages dédiés, nous sommes finalement en mesure de définir *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, et aucun des inconvénients.

Chapitre 4

Caractéristiques du langage hôte

Dans ce chapitre, nous allons examines les caractéristiques nécessaires d'un langage hôte pour qu'il puisse admettre un langage enchâssé.

Dans 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 fonction 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

Mais, il y a plusieurs types de macros. En particulier, les macros basées sur la substitution de texte, comme les macros du langage C, par exemple, sont inexploitables du point de vu d'un langage enchâssé. La raison est que les macros de ce type sont capable de ne pas respecter la structure de base du langage. Le système des macros C n'exige pas que les arguments d'une macro soient des expressions, des instructions ou d'autres éléments du langage, et le résultat de la macro-expansion ne l'est pas nécessairement non plus. Les macros C nécessitent alors une connaissance assez détaillé du fonctionnement interne de la macro, en particulier 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 ces 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 l'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. L'expansion nous donne la raison :

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

Ici, on voit que le plus grand entre les deux sera incrémenté deux fois et le résultat de l'affectation sera la valeur après une incrémentation. Il n'y a pas de façon 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 provoqué les créateurs du langage Java de 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 pour permettre un système de macros sans les difficultés d'un système basé sur la substitution de texte.

En fait, en Common Lisp, la représentation externe d'un programme sous la 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 immédiate que le programmeur Common Lisp fait automatiquement la traduction lui-même. Par exemple, lorsqu'il voit la suite des caractères :

```
(+ 3 4)
```

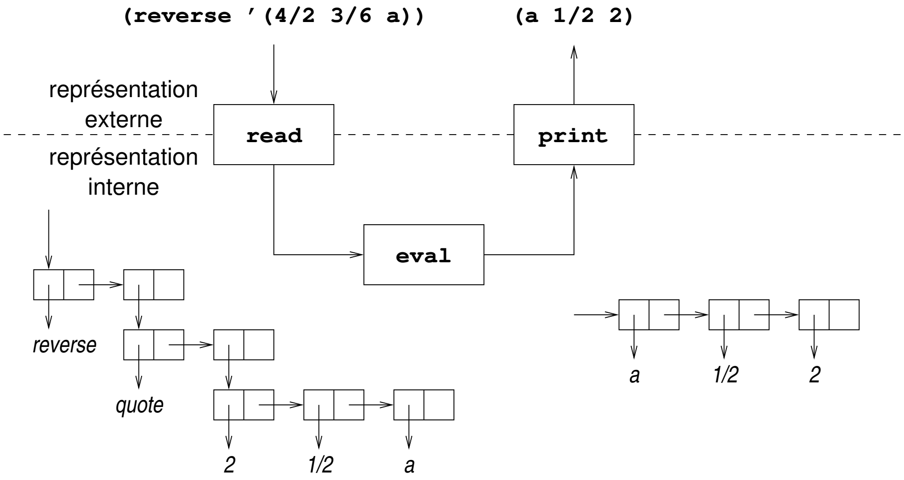

FIG. 4.1 – La boucle `read-eval-print`

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 définie 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 remplace 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 pour les 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 plus 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 est 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 un programme Common Lisp en entrée et qui produit un programme Common Lisp en sortie.

Pour envisager un langage enchâssés 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 être sous la forme d'un type de donnée de base du langage même,

Un tel langage sera 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 la forme d'une expression passée en argument et qui renvoie une autre expression comme valeur. Le compilateur, avant de compiler un appel macro (une expression commençant par le nom d'une macro), appelle une fonction pour calculer l'expansion de l'appel.

Chapitre 6

Macros de base

difficultés d'écriture. Capture de variables, multiple évaluation.

Chapitre 7

Deux styles de programmation mélangés

Dans la partie de la macro exécutée pendant l'expansion, le style est dense, plutôt fonctionnel mais pas forcement efficace, alors que dans la partie générée comme résultat de l'expansion, le style est totalement différent avec la performance comme objectif principal.

Troisième partie

Exemples de langages enchâssés

Chapitre 8

Une version préfixe du langage PostScript

8.1 Introduction

On considère une application ayant besoin de dessiner des figures construites d'une façon similaire à celle utilisée par PostScript. L'essence de la façon PostScript consiste à maintenir un environnement graphique contenant une transformation actuelle, un point de dessin actuel et un chemin actuel.

Le dessin se fait en deux phase. D'abord un chemin est construit. Puis le chemin est tracé. Les commandes de construction de chemin sont `moveto`, `lineto`, `arcto` et `curveto`. La commande `moveto` créé un nouveau sous chemin alors que les autres rajoutent un segment au sous chemin actuel.

Le chemin actuel est exprimé avec les coordonnés du système de repères physique (de l'imprimante ou de l'écran), alors que les commandes de construction de chemin sont exprimées avec les coordonnées du système de repères utilisateur actuel. C'est la transformation actuelle qui est chargée de la traduction entre les deux.

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

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

Notre petit langage sera appelé *prescript* car la notation sera préfixe alors que la notation de PostScript est postfixe.

8.2 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 : seul les commandes `moveto` et `lineto` seront implémentées.

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

8.2.1 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 (seul des segments de droites seront implémentés). Un segment commence implicitement dans le point de terminaison du segment précédent ou du point initial s'il s'agit du premier segment.

La commande `moveto` commence un nouveau sous chemin, qui 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 actuel. L'argument à la commande est le point de fin du segment, le point du début étant le point actuel avant l'exécution de la commande.

8.2.2 Traçage du chemin

Le chemin actuel est tracé grâce à la commande `stroke`. Cette commande ne prend pas d'arguments.

8.2.3 La transformation actuelle

Le chemin actuel contient des coordonnés physiques. Les commandes de construction du chemin prennent comme arguments des coordonnés dans un système de repères différent et modifiable par l'utilisateur. C'est la transformation actuelle qui est chargé de la traduction entre les deux systèmes de repères.

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

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

Une modification de la transformation actuelle influence les commandes de construction de chemin exécutées *après* la modification uniquement.

8.2.4 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 actuelle, le chemin actuel et le point actuel peuvent être localement modifiés sans effet global.

8.2.5 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 le petit langage s'intègre aussi 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)))
...
```

8.3 Le support commun

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

8.3.1 Le support X11

Ce code permet l'ouverture d'une fenêtre X11 pour que le résultat de la commande `stroke` puisse être vérifié.

Le code est exprimé en utilisant la bibliothèque CLX, qui est une implémentation en Common Lisp du protocole X11. Elle est 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 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 existe uniquement comme support du reste.

```
(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*))
```

8.3.2 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 pour illustrer la programmation orientée objets en Common Lisp.

Les objets de type `point` sont immuables. Cela évite la nécessité de copier le point actuel lors de l'exécution d'une commande `gsave`. Une modification du point actuel passe alors forcement par le remplacement du point actuel 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.

8.3.3 Transformation

Le code contenu de cette section permet de créer des transformation affines, de composer deux transformations, et de transformer un point par une transformation afin d'obtenir un autre point.

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 multiple pour récupérer les composants d'une transformation.

Les objets de type `transformation` sont immuables. Comme avec le point actuel, cela évite la nécessité de copier la transformation actuelle 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)))))
```

8.3.4 Chemin

Comme le point actuel et la transformation actuelle, le chemin actuel doit être immuable pour 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 la forme de liste de sous chemin. Chaque sous chemin est implémenté sous la forme de liste d'éléments. Le premier élément de chaque sous chemin est un point (le premier point du chemin), et 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 en sens 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 un valeur accumulée (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 renvoi 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 est 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))))
```

8.3.5 Segments

Seul le segment de droite est implémenté. Il faut imaginer ici une implémentation de chaque type de segment (arc, courbe).

On remarque surtout la méthode sur la fonction générique `stroke-one`. Cette méthode est appelée lorsque le deuxième argument à `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))
```

8.3.6 Environnement

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

Un objet de type `env` n'est pas immuable. Ces composants sont constamment modifiés par les commande 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)))
```

8.4 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 de ce type :

```
* (prescript-interpreter
```

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

*

Seul des constantes peuvent être utilisées comme arguments aux fonctions *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 de 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)))))
```

8.5 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 de 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 actuelle et le point de dessin actuel. 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 est pour 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 pour les 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 actuel, la variable `position` contient le point actuel, la variable `transformation` contient la transformation actuelle et la variable `path` contient le chemin actuel.

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 capture 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 de fonction. La raison est que l'environnement actuel 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))))))
```