# Université Montpellier II
# Maîtrise d'Informatique
# Module Langage, Evaluation, Compilation — 3 heures

R. Ducournau – M. Lafourcade

janvier 2003

*Documents autorisés : notes de cours, polys, pas de livre.*
*2 parties indépendantes, 6 questions notées sur 4 (une question bonus au choix).*
*NB. Un code Lisp schématique suffit à partir du moment où les points clés sont précisés.*

## 1 Optimisation des récursions terminales dans le code compilé

On se propose d'étudier une optimisation du code compilé pour les récursions terminales. Soit la fonction `foo` définie par :

```
(defun foo (x y z)
   (let ((a ---)
         (b ---))
        (...
           (foo --- --- ---)
          ...)))
```

On suppose que l'on a pu déterminer que l'appel récursif de `foo` est terminal. On va donc chercher à l'optimiser.

**Question 1**

Avant de voir comment optimiser l'appel récursif, on va d'abord examiner comment la génération de code standard et le code généré fonctionnent.

1. montrer l'état de la pile lors de l'appel récursif : pour que ce soit probant, faites figurer les paramètres de `foo` lors des 2 appels successifs, ainsi que les adresses de retour ;
2. montrer le code généré pour l'appel récursif.

**Question 2**

Une première façon consiste à traiter les récursions terminales lors de la génération de code. On suppose que l'on a pu déterminer, durant la génération de code de `foo`, que l'appel récursif est terminal. L'objectif va être de modifier le code généré de façon à ce que (1) l'appel récursif réutilise la pile occupée par le premier appel, (2) l'adresse de retour des deux appels soit la même et (3) l'appel s'effectue par un `JMP` au lieu d'un `JSR`.

1. montrer l'état de la pile lors de l'appel récursif ;
2. montrer le code généré pour l'appel récursif ;
3. donner le code Lisp schématique de la génération de code pour un appel terminal.

**Question 3**

Une autre façon de faire consiste à optimiser le code produit par la génération de code standard (question 1) de façon à le transformer dans le code de la question 2.

1. montrer comment on peut reconnaître un appel terminal à partir du code généré ;
2. décrire la transformation de code qu'il faut appliquer pour obtenir le code optimisé : en donner le code Lisp schématique.

## 2 Analyse statique et code intermédiaire

L'analyse statique regroupe un ensemble de techniques générales d'inspection du code source, à des fins variées : vérification de la correction, transformations source à source (par ex. expansion des macros), détection de forme particulières (par ex. récursions terminales, variables libres, fonctions non définies, etc.). L'analyse statique peut s'effectuer aussi bien pour du code à interpréter que pour du code à compiler : la génération de code elle-même constitue une analyse statique. Dans les deux cas, l'analyse statique peut produire des messages d'erreurs et une transformation dans un code intermédiaire qui tienne compte des informations détectées lors de l'analyse. Le principe de l'analyse statique est une « analyse par cas », similaire aussi bien à celle de la génération de code qu'à celle de la méta-évaluation.

**Question 4**

On spécifie le langage intermédiaire suivant, nommé PSIL, basé sur les mots-clés `:quote` pour les constantes, `:lambda` pour l'application de $\lambda$-expression, `:special-form` pour les formes spéciales (plus généralement pour les fonctions qui ont un traitement particulier dans le méta-évaluateur, autres que celles traitées par un des autres mots-clés), `:var` pour les variables, `:call` pour les appels de fonctions connues et `:unknown` pour les appels de fonctions pas encore connues.

1. Définir schématiquement la fonction `lisp2psil` qui prend en argument une expression LISP et retourne une expression PSIL suivant les spécifications suivantes :
   – toutes les macros sont expansées au fur et à mesure ;
   – toute expression LISP `expr` est remplacée par une expression PSIL `(keyword . expr)`, où le choix de `keyword` dépend du type de l'expression `expr`.
2. Indiquez brièvement comment le méta-évaluateur serait modifié pour méta-évaluer PSIL : quels mots-clés faudrait-il rajouter pour étendre PSIL ? Que faut-il faire en cas de définition de fonction (`defun`) ? Et pour la méta-évaluation du mot-clé `:unknown` ?
   Pensez-vous que ce sera plus efficace que méta-évaluer LISP ? Justifiez votre réponse ?

**Question 5**

Une *variable* est *libre* dans une expression si la variable figure dans l'expression sans avoir été introduite par un constructeur (`lambda`, `defun`, `let`, `labels`, etc.), interne à l'expression.

Modifier la fonction `lisp2psil` pour qu'elle détecte les variables libres : il faut lui ajouter un paramètre pour un environnement et elle affiche un message d'erreur (`warn`) lorsqu'elle rencontre une variable libre. Précisez ce que doit être l'environnement (une version simplifiée des environnements nécessaires à l'évaluation et à la génération de code). Traitez les cas de `lambda`, `defun`, `let`, `function` et `labels`.

**Question 6**

La notion d'appel récursif terminal s'élargit sans difficulté à celle d'appel terminal non récursif. L'appel de `bar` par `foo` est terminal si `foo` retourne la valeur retournée par `bar` sans effectuer dessus le moindre traitement :

```
(defun foo (x)
   (let ((a ---) (b ---))
        (... (bar --- --- --- ---) ...)))
```

L'analyse statique s'applique aussi aux appels terminaux et une extension de PSIL consiste à distinguer les appels terminaux, éventuellement récursifs, des appels enveloppés, par exemple en remplaçant le mot-clé `:call` par les mots-clés `:termcall` et `:envcall`.

Pour détecter un appel terminal, on rajoute un paramètre booléen à la fonction principale d'analyse statique (`lisp2psil`) et à certaines fonctions annexes : ce paramètre indique si la sous-expression courante constitue un appel terminal ou pas.

1. montrer comment ce paramètre est utilisé sur les cas du `if`, du `progn`, de la $\lambda$-expression et de l'appel de fonction ;
2. une modification très simple permet de reconnaître le cas où l'appel terminal est récursif : laquelle ?