# Université Montpellier II — Master d'Informatique
# FMIN 106 Compilation — 2 heures

R. Ducournau – M. Lafourcade

Janvier 2010

*Documents autorisés : notes de cours, polys, pas de livre.*
*Notation globale sur 23.*

Le point de départ de ce sujet est le langage intermédiaire du méta-évaluateur "threadé" (Chapitre 5 du polycopié "Compilation et Interprétation des Langages"). On suppose la transformation effectuée (par la fonction appelée `mtrans` dans le poly et `lisp2li` dans le cours) et on s'intéresse uniquement à des transformations, écrites en Lisp, d'expressions du langage intermédiaire.

On se restreindra à la partie du langage dont la syntaxe des expressions évaluables (`<expr-eval-li>`) est la suivante :

| | | |
|---|---|---|
| <expr-eval-li> | := | « (:**const .** » <expr> « ) » \| |
| | | « (:**var .** » <int> « ) » \| |
| | | « (:**if** » <expr-eval-li> <expr-eval-li> « **.** » <expr-eval-li> « ) » \| |
| | | « (:**progn** » <expr-eval-li> <expr-eval-li>+ « ) » \| |
| | | « (:**set-var** » <int> « **.** » <expr-eval-li> « ) » \| |
| | | « (:**mcall** » <symbol> <expr-eval-li>* « ) » \| |
| | | « (:**call** » <symbol> <expr-eval-li>* « ) » \| |
| | | « (:**unknown** » <expr-eval-lisp> « **.** » <env> « ) » |

Il y a donc dans l'ordre des mots-clés pour : les constantes et les variables, une forme conditionnelle, la séquence, l'affectation de variable, l'appel de fonction méta-définie, et de fonction prédéfinie, et enfin le cas des expressions encore inconnues au moment de la transformation.

Par rapport à Lisp, le langage est grandement simplifié : il n'y a plus de `let` (la transformation est censée les avoir fait disparaître, suivant le principe décrit dans le poly) et on ne s'intéresse pas ici à des constructions de plus haut niveau comme des fermetures, les fonctions locales ou les mots-clés `&rest` ou `&optional`. Il n'y a bien sûr plus de macros—sauf dans la forme :`unknown` si la macro a été définie après la transformation du code considéré.

La définition d'une fonction méta-définie, c'est-à-dire sa "valeur fonctionnelle", est constituée d'une paire contenant l'expression unique qui forme le corps de la fonction et un entier qui indique la taille de l'environnement à créer (ou la réservation de pile à faire) lors de l'appel.

## 1 Exemple de langage intermédiaire (sur 6)

On suppose de plus qu'il existe deux fonctions Lisp `get-defun` et `set-defun` qui permettent d'accéder à un symbole pour récupérer ou affecter sa "valeur fonctionnelle". Ainsi `(defun foo ...)` se traduit par `(set-defun 'foo <valeur fonctionnelle>)`. En contre-partie, on ne suppose l'existence d'aucun environnement fonctionnel.

**Question 1**

Définition de la fonction Fibonacci :

1. Ecrire d'abord en Lisp la définition de la fonction Fibonacci (`fibo`), dans sa forme récursive habituelle ;
2. puis en donner d'abord la valeur fonctionnelle telle qu'elle est produite par `lisp2li` dans le langage intermédiaire ;

3. l'intégrer ensuite dans le code du langage intermédiaire qui va affecter la valeur fonctionnelle et faire le `set-defun` ;
4. donner enfin la valeur fonctionnelle de `fibo` une fois que la fonction aura été utilisée.

## 2 Parcours d'arbres (sur 7)

La manipulation de langages formels n'est qu'une histoire de parcours d'arbres et d'analyse par cas.

**Question 2**

Définir la fonction Lisp `parcours-arbre-li` qui prend en paramètre une expression évaluable du langage intermédiaire et parcourt récursivement toutes ses sous-expressions évaluables (elles-mêmes du langage intermédiaire).

Cette fonction ne fait rien mais dans les questions suivantes on appliquera le même schéma pour faire différents traitements sur une expression : le parcours doit être *correct* (il ne traite que des expressions évaluables du langage intermédiaire) et *complet* (il les traite toutes). Il doit donc conrrespondre très exactement à la syntaxe de `<expr-eval-li>` ci-dessus.

**Question 3**

Pour donner de la consistance à cette fonctionnalité,

1. définir la fonction Lisp `parcours-arbre-li-example` qui
   - prend en entrée une expression évaluable du langage intermédiaire,
   - parcourt toutes ses sous-expressions évaluables,
   - en imprimant le mot-clé de l'expression et sa profondeur dans l'arbre,
   - et retourne le nombre d'expressions parcourues.
2. Faire tourner à la main la fonction `parcours-arbre-li-example` sur le corps de la fonction `fibo` de la question 1 en indiquant ce que l'appel de la fonction affiche.

## 3 Décompilation (sur 10)

On souhaite regénérer du code Lisp à partir du code intermédiaire, en définissant la fonction Lisp `li2lisp` qui, à partir de la définition d'une fonction en langage intermédiaire, produit la définition équivalente en Lisp standard (qui, par exemple, à partir de la définition en code intermédiaire de la fonction `fibo`, regénère le code Lisp d'une fonction équivalente à `fibo`).

**Question 4**

Spécifier et définir la fonction Lisp `li2lisp` en suivant les étapes suivantes :

1. On commence par définir une fonction `li2lisp-novar` qui ne traite que les expressions sans variables, c'est-à-dire les mots-clés `:const`, `:if`, `:call`, `:mcall` et `:progn` ;
2. On traite ensuite le cas des paramètres. Dès qu'il est question de variables, il faut un environnement, et pour générer les noms des paramètres, on utilisera la fonction `gensym`.
   On se restreint d'abord au cas des paramètres de la fonction (pas de variables locales) : compléter `li2lisp-novar` en une fonction `li2lisp-var` qui tienne compte de l'environnement et des mots-clés `:var` et `:set-var`, puis définir `li2lisp-fun` qui prend en argument le nom d'une fonction, crée un environnement et régénère la définition Lisp à partir de la définition en langage intermédiaire.
3. On souhaite ensuite tenir compte des variables locales. La définition d'une fonction méta-définie est maintenant constituée d'un triplet contenant le corps de la fonction, un entier $k$ qui indique la taille de l'environnement à créer, et le nombre $p$ de paramètres (avec $p \leq k$ bien entendu). Une variable locale est donc juste une place dans l'environnement, avec un `:set-var` pour l'affecter (il n'y a pas de mot-clé `:let`).
   Modifier les fonctions précédentes pour prendre en compte les variables locales.
4. Traiter enfin le cas de `:unknown`. On rappelle que, dans la syntaxe de `:unknown`, figure l'expression Lisp d'origine et un environnement qui sert à la fonction `lisp2li` et associe à chaque variable sa place dans l'environnement à générer à l'exécution.