IA41
Concepts fondamentaux en Intelligence Artificielle et langages dédiés

CM #8

Introduction au lambda-calcul Concepts fondamentaux de la programmation fonctionnelle avec LISP

Fabrice LAURI

utbm

Partie 1
Introduction au lambda-calcul

- Introduction

- Définitions : variables, applications, abstractions...

- Calcul sur les termes du λ-calcul

- Utilisation du λ-calcul : nombres entiers

Partie 2
Concepts fondamentaux de la programmation fonctionnelle avec LISP

- Introduction

- Mécanisme d'évaluation dans LISP

- Fonctions prédéfinies manipulant des listes

- Définition de fonctions utilisateurs

- Rappel sur la récursivité

Partie 1
Introduction au lambda-calcul

- Introduction

- Définitions : variables, applications, abstractions...

- Calcul sur les termes du λ-calcul

- Utilisation du λ-calcul : nombres entiers

Introduction

Crée dans les années 1930 par *Alonzo Church*.
Constitue la base formelle de la programmation fonctionnelle.

λ-calcul : formalise la notion de fonctions.

Fonctions mathématiques : définies en intension ou en extension.
λ-calcul : ne s'intéresse qu'aux fonctions définies via des
règles de calcul. Adapté à l'informatique (domaine du « fini »,
application de règles de calcul).

Fonction mathématique : domaine et co-domaine caractérisés.
Fonction du λ-calcul : domaine = co-domaine = termes du λ-calcul.

Partie 1
Introduction au lambda-calcul

- Introduction

- Définitions : variables, applications, abstractions...

- Calcul sur les termes du λ-calcul

- Utilisation du λ-calcul : nombres entiers

Définitions : notations

Termes du λ-calcul formés à partir de :
- symbole **λ**,
- **(** et **)**,
- **variables** x, y, z, ... contenues dans un ensemble dénombrable Σ
- symboles de **constantes** a, b, ... contenues dans un ensemble dénombrable Ω. Si Ω vide, on parle de λ-calcul pur.

Termes notés avec des lettres majuscules; constantes et variables avec des lettres minuscules.

Symbole ≡ dénote l'égalité syntaxique.

Définitions : variables, constantes, application et abstraction

Un terme est obtenu en appliquant un nombre fini de fois ces règles :

- **variable** et **constante** : tout symbole de variable et tout symbole de constante est un terme.
- **application** : Si A et B est un terme, alors **(AB)** est un terme.
- **abstraction** : Si x est une variable et A un terme, alors **(λx A)** est un terme.

(λx A) : fonction sans nom, dont le paramètre formel est x et le corps est A. L'abstraction (λx x) représente l'identité.

(AB) : application du terme A au terme B.

Définition : *curryfication*

Fonction de plusieurs variables, notée $\lambda x_1 x_2 x_3 ... x_k$ A, s'exprime aisément au moyen d'abstractions, fonctions à un seul argument.

Pour cela, utilisation de fonctions d'ordre supérieur : fonctions qui prennent en argument ou qui retournent d'autres fonctions.

Curryfication : procédé permettant de passer d'une fonction d'arité n à une fonction composée de fonctions toutes d'arité 1.

<u>Exemple :</u>

- f = (λxy A) est curryfiée en f' = (λx (λy A))
- f = (λfg (λx (f(gx)))) est curryfiée en f' = (λf (λg (λx (f(gx)))))

Peut s'appliquer à 1, 2 ou 3 arguments
retourne respectivement une fonction à 2, 1 ou sans arguments

Définitions : variables libres et liées

Variable libre
Toute variable ayant une occurrence non liée par un λ.

Variable liée
Toute variable ayant une occurrence liée par un λ.

Exemples :

- x est libre dans (λy (λz ((xy) z)))
- x est liée dans (λx (xx))
- x est à la fois liée et libre dans (λy (x (λx (yx))))

Combinateur, terme clos
Terme qui ne contient pas de variables libres.

Partie 1
Introduction au lambda-calcul

- Introduction

- Définitions : variables, applications, abstractions...

- Calcul sur les termes du λ-calcul

- Utilisation du λ-calcul : nombres entiers

Calcul sur les termes

Substitution d'un terme B à une variable libre dans un terme A

Soient 2 termes A et B et une variable x, [B/x]A, la substitution de B à chaque occurrence libre de x dans A, se définit par :

1. [B/x]x = B
2. Si A est un atome différent de x, [B/x]A = A
3. Si A est de la forme (CD), [B/x](CD) = ([B/x]C [B/x]D)
4. Si A est de la forme (λx C), [B/x]A = A
5. Si A est de la forme (λy C), y différent de x, alors :
   a) Si $y \notin$ VL(B) ou $x \notin$ VL(C),
      [B/x](λy C) = (λy [B/x] C)
   b) Si $y \in$ VL(B) et $x \in$ VL(C),
      [B/x](λy C) = (λz [B/x] [z/y] C)
      avec $z \notin$ VL((BC))

Exemples : [z/x] (λy x) et [z/x] (λz x)

avec VL(A) : ensemble des variables libres du terme A.

Calcul sur les termes

α-équivalence
Un terme A est α-équivalent à un terme D, noté $A \equiv_\alpha D$, ssi D a été obtenu à partir de A après application d'un nombre fini, éventuellement nul, de renommage de **variables liées**.

β-réduction
Modélisation du mécanisme de passage de paramètres.

Exemple : soit f(x) = x+x alors f(2) = [2/x] (x+x) = 2+2

Peut s'appliquer à chaque fois qu'une **abstraction** se voit présenter un argument dans une **application**.

Calcul sur les termes

β-redex
Tout terme A de la forme ((λx B) C). Ce terme peut se contracter pour former le terme [C/x] B, appelé *contractum*.
$A \blacktriangleright_{\beta}^{1} A'$: A se β-contracte en A'.
$A \blacktriangleright_{\beta} B$: A se β-réduit en B ssi B résulte de l'application d'un nombre fini de β-contractions à A.

Forme normale
Terme qui ne contient aucun β-redex.

Calcul sur les termes

Ordre de réduction (théorème de *Church-Rosser*)
Choisir le β-redex *maximal* le plus à gauche.

β-redex maximal
Soit un terme M. Un β-redex contenu dans M est maximal ssi il n'est contenu dans aucun autre β-redex de M.

Exemples de β-contractions :

- (((λx (λy x))a)b)
- ((λx (xx)) (λx (xx)))
- (((λx (λy x))a) ((λx (xx))(λx (xx))))

Condition d'arrêt de calcul via des β-réductions
Lorsque l'on aboutit à une forme normale (sans β-redex).
Il s'agira alors de la valeur finale de l'expression calculée.
Théorème de Church-Rosser comme méthode de calcul...

Partie 1
Introduction au lambda-calcul

- Introduction

- Définitions : variables, applications, abstractions...

- Calcul sur les termes du λ-calcul

- Utilisation du λ-calcul : nombres entiers

Représentation des nombres entiers en λ-calcul

Nombre n (*Church*) : fonction qui appliquera n fois son premier argument à son second argument.

Pour construire tous les entiers :
- définition du premier entier (0)

    0 ≡ (λf λx x)

- définition d'une fonction successeur

    S ≡ (λn λf λx ((nf) (fx)))

Successeur de 0,1,... :

(S 0) ≡ ((λn λf λx ((nf) (fx))) (λf λx x)) $\blacktriangleright_\beta$ *(λf λx (fx)) ≡ 1*

(S 1) ≡ *((λn λf λx ((nf) (fx))) (λf λx (fx)))* $\blacktriangleright_\beta$ *(λf λx (f (fx))) ≡ 2*

etc.

λ-calcul pur et λ-calcul appliqué

λ-calcul pur : trop de ressources (CPU+mémoire) pour l'utiliser tel quel en tant que langage de programmation fonctionnel.

Majorité des langages de programmation fonctionnel : basé sur un λ-calcul appliqué, c-à-d :
- définition de constante 0,1,2,... et +,-,/,*,...
- définition de règle de *δ-réduction*, donnant leur sens aux opérations +,-,/,*...

Ces règles (fonctions externes au λ-calcul), par exemple :

(+ m n) ➢ m+n
(* m n) ➢ m*n
...

permettraient de traiter plus efficacement les nombres entiers.

Partie 2
Concepts fondamentaux de la programmation fonctionnelle avec LISP

- Introduction

- Mécanisme d'évaluation dans LISP

- Fonctions prédéfinies manipulant des listes

- Définition de fonctions utilisateurs

- Rappel sur la récursivité

Introduction

Années 1930 : Création du λ-calcul (A. Church)
Années 1960 : Création du LISP (J. Mac Carthy)

LISP est l'acronyme de *LISt Processing*. LISP manipule des listes, et plus particulièrement des symboles.

Qu'est ce qu'une liste ?

$$liste := (e_1\ e_2\ ...\ e_n)$$

e_i est un élément, défini par :

$$e_i := atome \mid liste$$

avec :

$$atome := nombre \mid chaîne\ de\ caractère$$

Exemples d'atomes et de listes

- 123 : nombre

- « bonjour » : chaîne de caractères

- () ou *nil* : liste vide

- (ceci est une liste) : liste de longueur 4, profondeur 1

- (ceci (est (une)) autre (liste)) : liste de longueur 4, profondeur 3

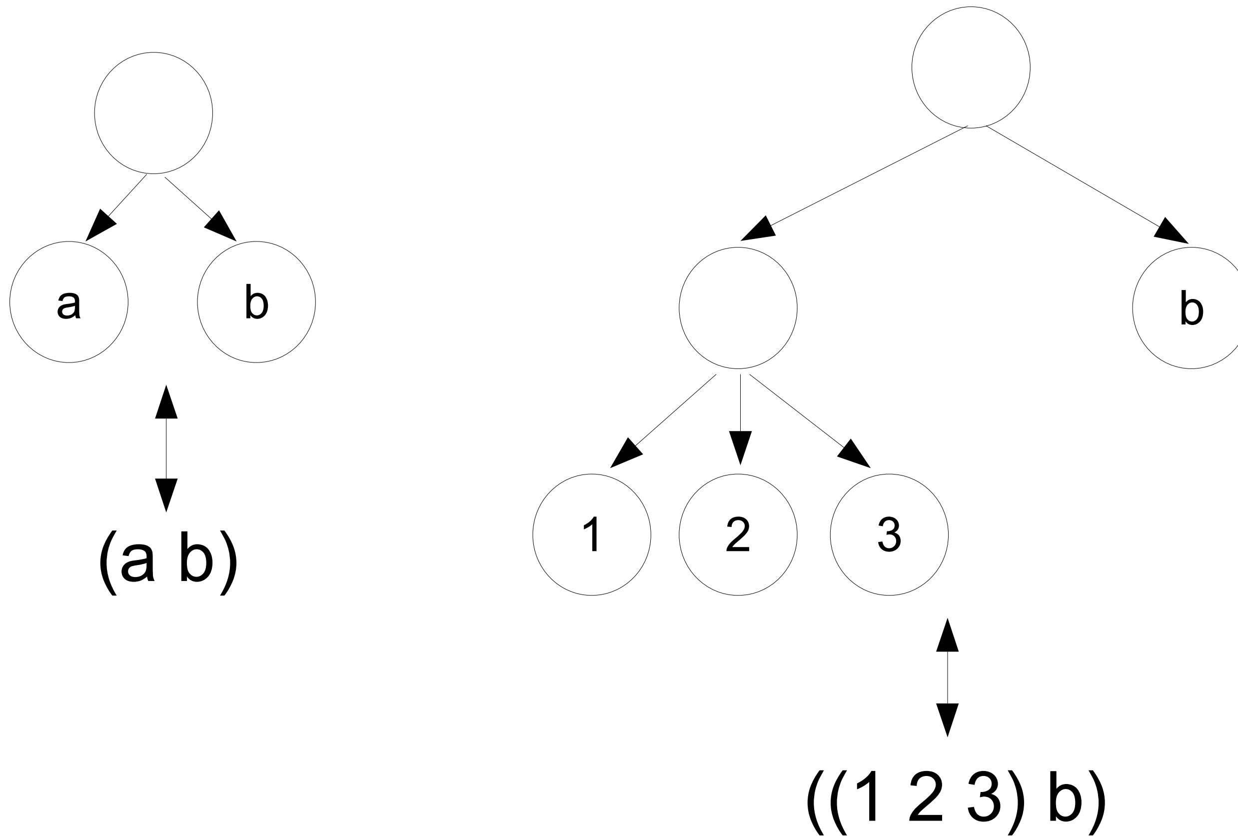
Un autre point de vue sur les listes : représentation sous forme d'arbres

Pour construire un arbre à partir d'une liste, on suppose que :

- une liste est représentée par un cercle vide
- les éléments d'une liste sont représentés par des cercles connectés directement au cercle de la liste
- la profondeur d'un élément est déterminée par le nombre de flèches que l'on rencontre pour l'atteindre dans l'arbre

La représentation d'une liste sous forme d'arbre permet à la fois de reconnaître graphiquement les éléments et leur profondeur.

Arbres et listes correspondantes



Le processus de transformation d'une liste en arbre est réversible. En particulier, les connaissances représentées dans un arbre peuvent également se traduire aisément sous la forme d'une liste.

Les listes spéciales : les formes

Les listes et les atomes sont les objets que vous pouvez manipuler en LISP.
Il existe des listes spéciales, les *formes*, qui indique à l'interpréteur LISP qu'il doit appliquer une fonction sur un ensemble d'arguments.

Une forme est définie par :

$$\textit{forme} := (\textit{nom-de-fonction}\ arg_1\ arg_2\ ...\ arg_n)$$

<u>Exemples :</u>

- (+ 1 2 3)
- (+ (* 5 2) 6)

...

Partie 2
Concepts fondamentaux de la programmation fonctionnelle avec LISP

- Introduction
- Mécanisme d'évaluation dans LISP
- Fonctions prédéfinies manipulant des listes
- Définition de fonctions utilisateurs
- Rappel sur la récursivité

Mécanisme d'évaluation de LISP (1/2)

Le mécanisme d'évaluation est systématique :

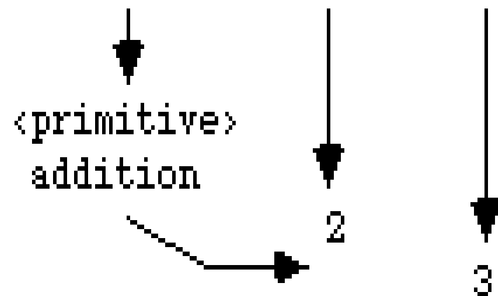
- si l'expression est un **nombre** ou une **chaîne de caractères**, cette expression est retournée comme valeur.

- si l'expression est une **liste** (commence par une '('), l'expression est considérée comme une fonction suivie d'arguments.
  Le premier élément de la liste (nom de la fonction) est évalué : il doit être une primitive ou une fonction utilisateur.
  Les éléments suivants de la liste sont évalués de gauche à droite et passés en argument à la fonction.

Ce mécanisme s'applique de façon récursive.

Mécanisme d'évaluation de LISP (2/2)

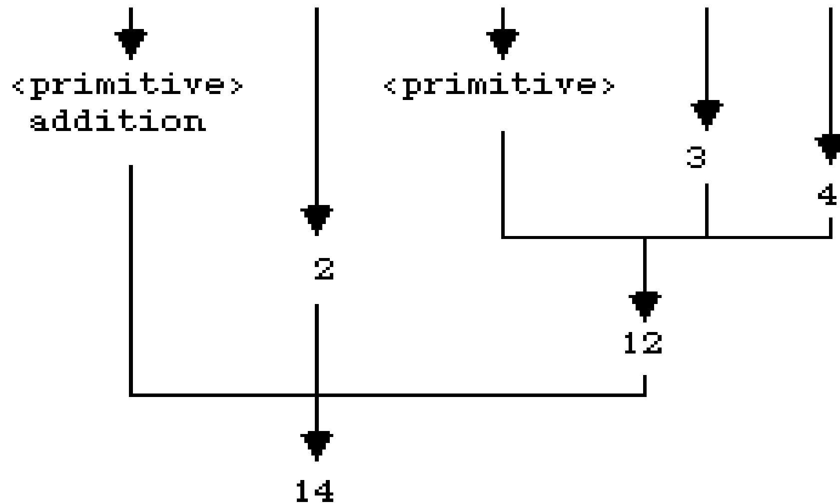
```
? ( +        2     3  )
```




```
; l'addition appliquée à 2 et 3 retourne 5
5
```

Exemple 1

```
? ( +        2      (*       3     4  ) )
```



Exemple 2

Partie 2
Concepts fondamentaux de la programmation fonctionnelle avec LISP

- Introduction

- Mécanisme d'évaluation dans LISP

- Fonctions prédéfinies manipulant des listes

- Définition de fonctions utilisateurs

- Rappel sur la récursivité

Les primitives sur les listes (1/3)

*(**car** <l>)* : tête (premier élément) de la liste <l>

*(**cdr** <l>)* : reste de la liste <l>

*(**cons** <e> <l>)* : ajout d'un élément <e> en tête de la liste <l>

*(**append** <l1> <l2>)* : concaténation de deux listes <l1> et <l2>

*(**list** <e1> ... <en>)* : construction d'une liste à partir d'éléments

*(**length** <l>)* : nombre d'éléments de la liste <l>

Les primitives sur les listes (2/3)

*(**quote** <l>)* : bloque l'évaluation de la liste <l>

Exemples :
(quote (a b c)) → (a b c)
(quote (+ 1 3 5)) → (+ 1 3 5)
'(+ 1 3 5) → (+ 1 3 5)

*(**eval** <l>)* : force l'évaluation de la liste <l>.

Exemples :
(cons '+ '(1 3 5)) → (+ 1 3 5)
→ *liste représentant une **donnée***

(eval (cons '+ '(1 3 5))) → 9
→ *évaluation des données : programme*

Les primitives sur les listes (3/3)

*(**null** <l>)* : argument <l> est-il une liste vide ?

*(**atom** <a>)* : argument <a> est-il un atome ?

*(**consp** <l>)* : argument <l> est-il une liste **non vide** ?

*(**endp** <l>)* : la liste <l> est-elle vide ?

*(**member** <e> <l>)* : argument <e> est-il membre de la liste <l> ?

Partie 2
Concepts fondamentaux de la programmation fonctionnelle avec LISP

- Introduction

- Mécanisme d'évaluation dans LISP

- Fonctions prédéfinies manipulant des listes

- Définition de fonctions utilisateurs

- Rappel sur la récursivité

Définition de fonctions utilisateurs

Il est possible d'automatiser des traitements symboliques spécifiques en définissant des fonctions utilisateurs.
La fonction ***defun*** permet de définir une nouvelle fonction.

La définition syntaxique de *defun* est :

(*defun* nom-de-fonction ({var_1 var_2 ... var_n}) corps-de-fonction)

noms des paramètres

suite d'appels de fonctions

Par exemple, la fonction *quatrieme,* qui retourne le 4ème élément d'une liste, pourra être définie ainsi :

(*defun* quatrieme (l) (car (cdr (cdr (cdr l)))))

Partie 2
Concepts fondamentaux de la programmation fonctionnelle avec LISP

- Introduction
- Mécanisme d'évaluation dans LISP
- Fonctions prédéfinies manipulant des listes
- Définition de fonctions utilisateurs
- Rappel sur la récursivité

Définition de la récursivité

La récursivité est la possibilité de faire apparaître dans la définition d'une entité une référence à elle-même. Dans ce cas, on dit que l'entité possède une définition récursive ou qu'elle est intrinsèquement récursive.

En programmation, on distingue deux types d'entités récursives :

- les fonctions récursives
    - fonction factorielle, fonction puissance
    - fonction de *Fibonacci*

- les structures de données récursives
    - les nombres
    - les listes
    - les arbres
    - les objets fractals...

Approche pour la construction d'algorithmes récursifs (1/2)

Un problème se prête bien à l'analyse récursive lorsqu'il peut être décomposé en sous-problèmes de taille plus petite et de même nature que le problème initial.

L'analyse récursive est constituée de trois étapes :

1) paramétrage du problème

2) recherche d'un cas trivial et de sa solution

3) décomposition du cas général

Approche pour la construction d'algorithmes récursifs (2/2)

1) Paramétrage du problème

Il s'agit d'identifier tous les paramètres du problème, en particulier ceux dont la taille décroît à chaque appel récursif.

2) Recherche d'un cas trivial et de sa solution

Un cas trivial est un sous-problème qui peut être résolu **SANS appel récursif**. Il correspond souvent au cas où la taille est nulle.

3) Décomposition du cas général

Cette étape a pour but de ramener le problème donné à l'instant t vers un ou plusieurs sous-problèmes que l'on suppose déjà traités à des instants précédents et de taille plus petite.

Application de l'approche : définition de la fonction factorielle

Par exemple, définir la fonction factorielle revient à spécifier les trois étapes suivantes :

1) Paramétrage (profil) de la fonction

$$factorielle : N \rightarrow N$$

2) Cas trivial

$$Pour\ n < 2,\ factorielle(\ n\ ) = 1$$

3) Cas général (appel récursif)

$$Pour\ n >= 2,\ factorielle(\ n\ ) = n * factorielle(\ n-1\ )$$

Notations pour l'écriture d'une fonction : exemple avec la fonction factorielle

1. Profil :

factorielle : N → N

2. Définition formelle de factorielle(n) :

$\{ n < 2 \} \quad \rightarrow factorielle(n) = 1$
$\{ n >= 2 \} \quad \rightarrow factorielle(n) = n * factorielle(n-1)$

3. Jeu d'essais :

factorielle(3) = *3 * factorielle(2)*
= *3 * 2 * factorielle(1)*
= *3 * 2 * 1*
= *6* (voir page suivante)

Exécution d'une fonction récursive : exemple avec la fonction factorielle

factorielle(3) → 3 * **factorielle(2)**

(1) Appel récursif

factorielle(2) → 2 * **factorielle(1)**

(2) Appel récursif

factorielle(1) → 1

Exécution d'une fonction récursive : exemple avec la fonction factorielle

(3) ↑ Valeur = 6

factorielle(3) → 3 * **factorielle(2)**

(2) ↑ Valeur = 2

factorielle(2) → 2 * **factorielle(1)**

(1) ↑ Valeur = 1

factorielle(1) → 1

Définition de la fonction factorielle en LISP

```
(defun
    fact (n)
    (if
        (< n 2)
        1
        (* n (fact (- n 1)))))
```

avec :

*(**if** <condition> <action-then> <action$_1$-else>...<action$_N$-else>)*
si <condition> est évaluée à *non-nil*, **if** retourne l'évaluation de <action-then>.
Si <condition> est *nil*, les expressions <action$_i$-else> sont évaluées séquentiellement et la valeur de <action$_N$-else> est retournée.