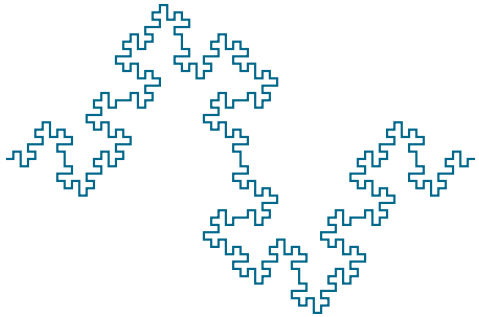
HLIN403 – Programmation Applicative

Identificateurs, Fonctions, Premières Structures de contrôle

Christophe Dony – Annie Chateau
Université Montpellier – Faculté des Sciences



Introduction

LAMBDA-CALCUL

Le **lambda-calcul** est un système formel (Alonzo Church 1932)

Langage de programmation théorique [1] qui permet de modéliser les fonctions calculables, récursives ou non, et leur application à des arguments.

Le vocabulaire et les principes d'évaluation des expressions en *Lisp* ou *Scheme* sont hérités du lambda calcul.

1. Jean-Louis Krivine, Lambda-Calcul, types et modèles, Masson 1991

LAMBDA-CALCUL

Les expressions du lambda-calcul sont nommées **lambda-expressions** ou *lambda-termes*, elles sont de trois sortes : variables, applications, abstractions.

* **variable** : équivalent des variables en mathématique (x, y ...)

* **application** : notée “uv”, où u et v sont des lambda-termes représente l’application d’une fonction `u` à un argument `v` ;

* **abstraction** : notée “$\lambda x.v$” ou x est une variable et v un lambda-terme, représente une fonction, donc l’abstraction d’une expression à un ensemble de valeurs possibles. Ainsi, la fonction f qui prend en paramètre le lambda-terme x et lui ajoute 2 (c’est-à-dire la fonction $f : x \mapsto x + 2$) sera dénotée en lambda-calcul par l’expression $\lambda x.(x + 2)$.

LAMBDA-CALCUL

Le lambda-calcul permet le raisonnement formel sur les calculs réalisés à base de fonctions grâce aux deux opérations de transformation suivantes :

- **alpha-conversion** :
  $\lambda x.xv \equiv \lambda y.yv$
- **beta-réduction** :
  $(\lambda$ x.xv) a $\rightarrow$ av
  $(\lambda x.(x+1))3 \rightarrow 3+1$

LAMBDA-CALCUL

Ce sont les mêmes constructions et opérations de transformations qui sont utilisés dans les langages de programmation actuels.

Nous retrouvons en Scheme les concepts de variable, fonction, application de fonctions et le calcul de la valeur des expressions par des beta-réductions successives.

Identificateurs

identificateur, en informatique, nom donné à un couple "emplacementMémoire-valeur".

Selon la façon dont un identificateur est utilisé dans un programme, il dénote soit l'emplacement soit la valeur.

Dans l'expression `(define pi 3.1416)`, l'identificateur `pi` dénote un emplacement en mémoire.

Dans l'expression `(* pi 2)`, l'identificateur `pi` dénote la valeur rangée dans l'emplacement en mémoire nommé *pi.*

Identificateurs

```
(define id v)
```

`define` est une structure de contrôle (voir plus loin) du langage *Scheme* qui permet de ranger la valeur `v` dans un emplacement mémoire (dont l'adresse est choisie par l'interpréteur et inaccessible au programmeur) nommé `id`.

Evaluation d'un identificateur

Soit *contenu* la fonction qui donne le contenu d'un emplacement mémoire et *caseMémoire* la fonction qui donne l'emplacement associé à un identificateur alors :

```
val (ident) = contenu (caseMemoire (ident))
```

Il en résulte que :

```
> (* pi 2)
6.28...
```

Erreurs liées à l'évaluation des identificateurs : *reference to undefined identifier :*

Environnement

Environnement : ensemble des liaisons “identificateur-valeur” définies en un point d’un programme.

Un environnement est associé à toute exécution de fonction (les règles de masquage et d’héritage entre environnements sont étudiées plus loin).

L’environnement associé à l’application de la fonction g ci-dessous est `((x 4) (y 2))`.

```
>(define x 1)
>(define y 2)
>(define g (lambda (x) (+ x y)))
>(g 4)
6
```

Abstraction

Une fonction définie par un programmeur est une abstraction du lambda-calcul et prend la forme syntaxique suivante :
`(lambda(param1 paramN) corps)`

Une fonction possède des paramètres formels en nombre quelconque et un corps qui est une S-expression.

L'extension à un nombre quelconque de paramètres est obtenue par un procédé dit de "Curryfication" - voir ouvrages sur le lambda-calcul.

Abstraction

La représentation interne d'une abstraction est gérée par l'interpréteur du langage, elle devient un élément du type prédéfini `procédure`.

```
> (lambda (x) (+ x 1))
#<procedure:15:2>
```

On distingue ainsi le texte d'une fonction (texte de programme) et sa représentation interne (codée, en machine).

Ecrire un interprète d'un langage, c'est aussi choisir la représentation interne des fonctions créées par l'utilisateur

Application

Appliquer une fonction à des arguments signifie exécuter le corps de la fonction dans un environnement ou les paramètres formels sont liés à des valeurs.

Les valeurs sont les valeurs des **arguments** passés à l'appel de la fonction.

Syntaxe : `(fonction argument1 ... argumentN)`

```
> ((lambda (x) (+ x 1)) 3)
4
> ((lambda (x) (* x x)) (+ 2 3))
25
```

Pas à pas

```
--> ((lambda (x) (+ x 1)) 3)
--> (lambda (x) (+ x 1))
<-- #<procedure:15:2>
--> 3
<-- 3
--> (+ x 1)
--> +
<-- #<primitive:+>
--> x
<-- 3
--> 1
<-- 1
<-- 4
<-- 4
```

Identificateurs et fonctions

Il est possible de dénoter une fonction par un identificateur en utilisant la structure de contrôle `define`.

```
(define f (lambda(x) (+ x 1)))
> (f 3)
4
> (define carre (lambda (x) (* x x)))
> (carre 5)
25
> (define x 10)
> (carre 5)
25
```

Identificateurs et fonctions

parametre formel : identificateur dénotant, uniquement dans le corps de la fonction (portée) et durant l'exécution de la fonction (durée de vie), la valeur passée en argument au moment de l'appel.

durée de vie d'un identificateur : intervalle de temps pendant lequel un identificateur est défini.

portée d'un identificateur : Zone du texte d'un programme où un identificateur est défini.

Séquence d'instructions

Structure de contrôle : fonction dont l'interprétation nécessite des règles spécifiques.

Donné pour info, inutile en programmation sans effets de bord, mais nécessaire par exemple pour réaliser des affichages.

```
(begin
i1
i2
...
in)
```

Séquence d'instructions

Il y a un *begin* implicite dans tout corps de fonction.

```
(lambda () (display "la valeur de (+ 3 2) est : ") (+
3 2))
```

évaluation d'une séquence :

`val ( (begin inst1 ... instN expr) ) = val (expr)` avec comme effet de bord, `eval(inst1), eval(instN)`

Conditionnelles

```
(define (abs x)
  (if (< x 0) (- 0 x) x))

(define (abs x)
  (cond ((> x 0) x)
   ((= x 0) 0)
   (t (- 0 x))))

(define (>= x y)
  (or (= x y) (> x y)))
```

Evaluation d'une conditionnelle de type "if" :

`val ( (if test av af) ) = si` $val(test) = vrai$ `alors` $val(av)$ `sinon` $val(af)$

Nombres

tests : integer ? rational ? real ? zero ? ; odd ? even ?

comparaisons `< > <= ...`

```
> (< 3 4)
#t
> (rational? 3/4)
#t > (+ 3+4i 1-i)
4+3i
```

Caractères

constantes littérales : `#\a #\b`

comparaison : `(char<? #\a #\b) (char-ci<? #\a #\b)`

tests : `char? char-numeric?`

transformation : `char-upcase`.

Chaînes de caractères (Strings)

constantes littérales : `"abcd"`

comparaison : `(string<? "ab" "ba")`

tests : `(string=? "ab" "ab")`

accès :

```
(substring s 0 1)
(string-ref s index)
(string->number s)
(string-length s)
```

Chaînes de caractères (Strings)

Exemple, fonction rendant le dernier caractère d'une chaîne :

```
(define lastchar
  (lambda (s)
    (string-ref s (- (string-length s) 1))))
```