Université Montpellier-II

UFR des Sciences - Département Informatique - Licence Informatique

Programmation Applicative et Récursive

Cours No 3 : Identificateurs, Fonctions, Premières Structures de contrôle.

Notes de cours

5 Lambda-calcul

Le **lambda-calcul** est un système formel (Alonzo Church 1932), (langage de programmation théorique[a]) qui permet de modéliser les fonctions calculables, récursives ou non, et leur application à des arguments. Le vocabulaire relatif et les principes d'évaluation des expressions en *Lisp* ou *Scheme* sont hérités du lambda calcul.

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

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 1 (c'est-à-dire la fonction $f : x \to x + 2$) sera dénotée en lambda-calcul par l'expression $\lambda x.(x + 2)$.

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

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

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.

6 Identificateurs

6.1 Définition

identificateur 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 [a].

Dans l'expression `(define pi 3.14116)`, l'identificateur `pi` dénote un emplacement en mémoire.

`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`.

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

a. voir termes L-Value (emplacement) et R-Value (contenu).

6.2 Evaluation d'un indentificateur

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 pour tout identificateur `ident` :

$$val(ident) = contenu(caseMemoire(ident))$$

Il en résulte que :

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

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

6.3 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))`.

```
1 >(define x 1)
2 >(define y 2)
3 >(define g (lambda (x) (+ x y)))
4 >(g 4)
5 6
```

7 Définition de fonctions

7.1 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 [a] et un corps qui est est une S-expressions.

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

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

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

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

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

7.2 Application

L'application de toute fonction est identique à l'application des fonctions prédéfinies à ceci près que les premières ne sont pas nécessairement associées à un identificateur.

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

Pas à pas.

```
1  ——> ((lambda (x) (+ x 1)) 3)
2  ——> (lambda (x) (+ x 1))
3  <—— #<procedure:15:2>
4  ——> 3
5  <—— 3
6  ——> (+ x 1)
7  ——> +
8  <—— \#<primitive:+>
9  ——> x
10 <—— 3
11 ——> 1
12 <—— 1
13 <—— 4
14 <—— 4
```

7.3 Identificateurs

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

Ceci permet de conserver la valeur d'un calcul déjà réalisé pour éviter d'avoir à le refaire.

```
1 (define f (lambda(x) (+ x 1)))
2 > (f 3)
3 4
4 > (define carre (lambda (x) (* x x)))
5 > (carre 5)
6 25
7 > (define x 10)
8 > (carre 5)
9 25
```

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.

8 Premières Structures de contrôle

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

8.1 Séquence d'instructions

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

```
1 (begin
2     i1
3     i2
4     ...
5     in)
```

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

```
1 (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 : $eval(inst1), ..., eval(instN)$

8.2 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 :

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

9 Fonctions de base sur les types prédéfinis

9.1 Nombres

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

comparaisons `< > <= ...`

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

9.2 Caractères

constantes littérales : `#\a #\b`

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

tests : `char? char-numeric?`

transformation : `char-upcase`.

9.3 Chaînes de caractères (Strings)

constantes littérales : `"abcd"`

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

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

accès :

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

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

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

10 Blocs, évaluations en liaison lexicale

10.1 Blocs

Un bloc (concept introduit par algol) est un séquence d'instructions à l'exécution de laquelle est associée un environnement propre.

En *Scheme*, toute fonction définit implitement un bloc.

```
1 > (define (f x) (* x x))
2 > (f 2)
3 = 3
```

La structure de contrôle **let** définit également un bloc.

Syntaxe :

```
1  (let ( <liaison>* ) expression*)

3  liaison ::= (identificateur s-expression)
```

Exemple :

```
1  > (let ((x (+ 2 3)) (y (+ 3 4)))
2       (* x y))
3  = 35
```

10.2 Evaluation d'un "let"

.

Soit i une instruction de la forme `(let ((v1 expr1) ... (vn exprN)) expr)`, on a $val(i) = val(expr)$, avec *expr* évalué dans l'environnement du let augmenté des liaisons v1 à $val(expr1)$, ..., vn à $val(expnN)$.

10.3 Bloc et environnement

L'environnement associé à un bloc de code est initialisé au moment de l'exécution du bloc, (exécution du corps d'une fonction ou du corps d'un *let*.

Pour une fonction, les paramètres formels sont liés aux arguments dans l'environnement associé au corps de la fonction.

define : est une structure de contrôle permettant d'ajouter un identificateur ou d'en modifier un dans l'environnement courant.

Modularité : structuration d'un programme en espaces de nommage.

Espace de nommage : partie d'un programme dotée d'un environnement propre.

Un bloc est un espace de nommage. Il est possible d'utiliser sans ambigüité le même identificateur dans des blocs différents.

```
1 (define x 12)
2 (define (carre x) (* x x))
3 (define (cube x) (* x x x))
```

10.4 Chaînage des environnements

Environnement fils : Un environnement E2 peut être chainé à un environnement E1, on dit que E2 étends E1, (on dit aussi que E2 est fils de E1). E2 hérite alors de toutes les liaisons de E1 et y ajoute les siennes propres (avec possibilité de masquage).

La façon dont les environnements sont chaînés définit la portée des identificateurs.

10.5 Portée lexicale

Le chaînage est dit statique ou lexical quand un environnement d'un bloc est créé comme le fils de l'environnement du bloc englobant lexicalement (inclusion textuelle entre zones de texte).

La portée des identificateurs est alors lexicale, i.e. un identificateur est défini dans toutes les régions de programme englobée par le bloc ou se trouve l'instruction réalisant la liaison.

Scheme obéit à ce modèle (voir exemples) ainsi que la pluspart des langages.

10.6 Portée dynamique

Le chaînage entre environnements est dynamique, et la portée des identificateurs également, quand l'environnement associé à une fonction devient, à chaque exécution, le fils de celui associé à la fonction appelante.

La portée dynamique a presque disparue ...

10.7 Environnement lié à la boucle toplevel

Environnement global de scheme : environnement dans lequel sont définis tous les identificateurs liés aux fonctions prédéfinies. Tout autre environnement est indirectement (fermeture transitive) le fils, de cet environnement global.

Environnement initial : environnement affecté à la boucle toplevel i.e. dans lequel sont interprétées les expressions entrées au toplevel. Cet environnement est un fils de l'environnement global. Tout bloc définit "au toplevel" créé un environnement fils de cet environnement initial.

NB : les fonctions définies dans l'éditeur sont conceptuellement définies au toplevel.

10.8 exemples

Variable définie ou indéfinie

```
1  > (define x 22) ;; liaison de x à 22 dans l'env. courant
2  > (+ x 1) ;; une expression E
3  = 23 ;; ayant une valeur dans l'env courant
4  > (+ i 1) ;; une autre expression n'ayant pas de valeur
5  erreur : reference to undefined identifier: i
```

Environnement fils

```
1  > (define x 10) ;; enrichir l'environnement courant
2  > (let ((y 30) (z 40)) ;; créer un environnement fils
3       (+ x y z)) ;; et y faire un calcul
4  = 80
5  > x
6  = 10
7  > y
8  erreur : reference to undefined identifier: z
```

Attention aux variables libres

```
1  (define x 1)

3  (define f (lambda (y) (g)))

5  (define g (lambda () y))

7  > (f 3)
8  *** reference to undefined identifier: y
```

```
1  (define x 1)
2  (define (f x) (g 2))
3  (define (g y) (+ x y)) ;; Une fonction avec une variable libre
4  > (f 5)
5  = 3 (vaudrait 7 avec portée dynamique)
```

Utilisation préférentielle de variables locales

```
1  > (define (nb-racine a b c)
2      (define delta (- (carre b) (* 4 a c))
3      (if (= delta 0) ...)
4    )
5  > delta
6  erreur : reference to undefined identifier: delta
```

```
1  (define (nb-racine a b c)
2      (let ((delta (- (carre b) (* 4 a c))))
3        (if (= delta 0)
4            ...)
5  )
```

Espaces de nommage

```
9  (define carre (lambda (x) (* x x)))

11 (define (sommeCarres x y) ; syntaxe simplifiée
12     (+ (carre x) (carre y)))

14 > (sommeCarres (sin 60) (cos 60))
15 = 1
```

Combinaisons plus complexes

```
16 (define x 1)

18 (define f (lambda() x))

20 (define g (let ((x 2)) (lambda() (+ x 2))))

22 (define h
23   (lambda (uneFonction)
24     (let ((x 3))
25       (apply uneFonction ()))))

27 > (f)
28 1
29 > (g)
30 4
31 > (define x 5)
32 > (f)
33 5
```

```
> (g)
4
> (h g)
4
```