Module I2

Informatique
Deug MIAS - Nîmes
1er semestre 97/98

Une Introduction à la Programmation

- ou comprendre un outil de calcul afin d'en mieux user -

Phil. REITZ

adr : LIRMM
161, rue Ada
34392 Montpellier cedex 5
tél. : 04 67 41 85 35
fax : 04 67 41 85 00
mél. : reitz@lirmm.fr

Organisation générale du module

Emploi du temps prévisionnel

module I2 = 56h éqTD = 18h C + 20h TD + 14h TP

Date	8h15-10h15	10h30-12h30
26 septembre	2h C	2h TD
3 octobre	2h C	2h TD
10 octobre	2h C	2h TD
17 octobre	2h C	2h TD
24 octobre	2h C	2h TD
31 octobre	2h C	2h TD
7 novembre	2h C	2h TD
14 novembre	2h C	2h TD
21 novembre	2h C	2h TP
28 novembre	2h TD	2h TP
5 décembre	2h TD	2h TP
12 décembre	2h TP	2h TP
19 décembre	2h TP	2h TP

Evaluation

un examen (3h) en janvier, tous documents autorisés

Supports

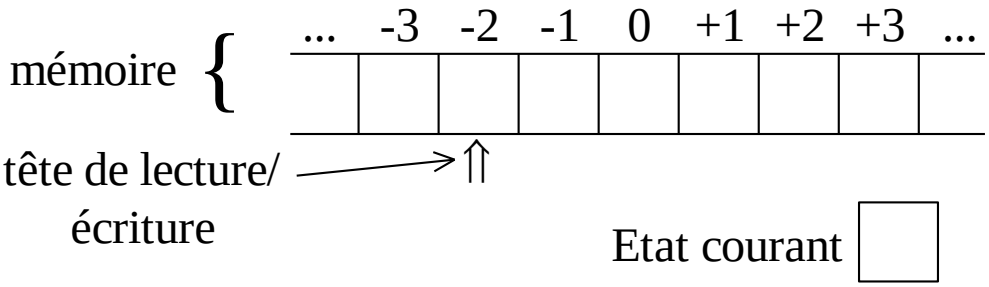
une copie des transparents du cours

corrections des TD, TP, et examens précédents

quelques bouquins de référence

Une illustration

Présentation d'un mécanisme de calcul



la mémoire :

- une bande infinie de ***cases mémoires*** numérotées
- chaque case mémoire contient un ***symbole*** pris dans un ensemble fini (***alphabet***)
- une ***tête*** de lecture/écriture repère une case mémoire
- cette tête peut se déplacer

A chaque instant, la machine est complètement caractérisée par sa ***configuration*** :

- le contenu de sa mémoire (i.e. pour chaque case, quel symbole)
- la position de la tête de lecture/écriture
- son ***état courant*** (un nombre entier naturel)

Faire un ***calcul***, c'est transformer la configuration courante.

L'évolution d'une configuration s'appuie sur une ***table de transition*** :

état courant	symbole lu par la tête	$\Rightarrow$	prochain état courant	symbole à écrire par la tête	déplacement de la tête

Une illustration

principe de fonctionnement :

la configuration courante est telle que :

- la tête de lecture/écriture pointe sur une case *c*
- le contenu de la case *c* est le symbole *s*
- l'état courant est *e*

Effectuer un ***pas de calcul***, c'est :

[1] rechercher dans la table de transition la ligne telle que l'état courant et le symbole lu soient ceux de la configuration courante (*e* et *s*).

supposons que cette ligne soit de la forme :

état courant	**symbole lu par la tête**	⇒	**prochain état courant**	**symbole à écrire par la tête**	**déplacement de la tête**
e	*s*		*e'*	*s'*	*d*

[2] transformer la configuration de la machine selon les indications de la table de transition :

- l'état courant devient *e'*
- le symbole *s* de la case *c* est remplacé par le symbole *s'*
- la position de la tête est déplacée de *d* cases

[3] si le déplacement *d* est non nul, revenir au point [1] (pas de calcul suivant), en travaillant avec la nouvelle configuration obtenue au point [2]

sinon (*d* est nul) s'arrêter ; le calcul est terminé.

Une illustration

Un exemple de fonctionnement

alphabet = { ● × ∴ }

table de transition :

état courant	symbole lu	prochain état	nouveau symbole	déplacement
1	×	2	×	+1
2	●	2	●	+1
2	×	3	●	+1
3	∴	3	×	-1
3	●	3	●	-1
3	×	3	×	0

configuration initiale :

...	-3	-2	-1	0	+1	+2	+3	...
∴	∴	×	●	●	●	×	∴	∴

⇑

Etat courant | 1 |

configuration suivante (après 1 pas de calcul) :

...	-3	-2	-1	0	+1	+2	+3	...
∴	∴	×	●	●	●	×	∴	∴

⇑

Etat courant | 2 |

configuration finale (après 10 pas de calcul) :

...	-3	-2	-1	0	+1	+2	+3	...
∴	∴	×	●	●	●	●	×	∴

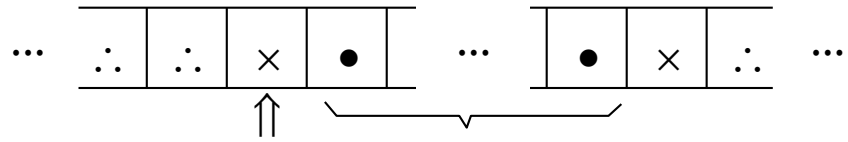
⇑

Etat courant | 3 |

Une illustration

Un exemple de fonctionnement : interprétation

La mémoire contient une représentation d'un nombre entier naturel *n* :



n fois

La tête est supposée se trouver sur le × de gauche, au début et à la fin du calcul

Le programme a pour objectif de transformer *n* en *n*+1 :

le × de droite est transformé en ● ×

Si *n* est le nombre codé en mémoire, ce programme nécessite 2(*n*+2) pas de calcul.

Il existe un programme capable de résoudre le même problème en un nombre fixe de pas de calcul.

Une illustration : premier bilan

Quelles leçons tirer de cet exemple

- Un programme permet de contrôler un mécanisme afin de lui faire résoudre un problème
- Cette résolution passe par un codage du problème, de sa solution, et de toute étape intermédiaire
- Il n'existe pas forcément un seul programme pour résoudre un même problème
- Il est possible de comparer l'efficacité les programmes (étude de la ***complexité***)
- Il est possible de prouver que, quelque soit n en début de calcul, le résultat final est toujours $n+1$ (***preuve*** de programmes)

  le programme est donc bien une réalisation de la fonction mathématique f suivante :

$$f(n) = n+1$$

Remarque

Le mécanisme de calcul présenté est la machine (abstraite) de Turing.

Terminologie

Objectif de l'informatique

étant donnés :

- une machine de calcul
- une fonction spécifiant ce que doit produire un programme

alors :

- construire un programme contrôlant la machine afin d'être en mesure d'interpréter son comportement comme une réalisation de la fonction

Les fonctions

Un programme est l'incarnation d'une ***fonction*** $f : E \rightarrow S$

- si $e \in E$, alors $s=f(e)$ est unique
- il peut exister e et e' tels que $s = f(e) = f(e')$

En informatique, E et S doivent être ***dénombrables***

X dénombrable = ses éléments peuvent être numérotés

= il existe une bijection entre X et un sous-ensemble de N

Terminologie

Les modèles de calcul

Un ***modèle*** (ou ***machine***) ***de calcul*** consiste en la donnée de :

- un ***substrat*** permettant de représenter ***problèmes*** et ***solutions***
- un ensemble d'***actions*** élémentaires capables de modifier ce substrat
- un ***système d'agencement*** et d'***exécution*** de ces actions

Les modèles de calcul les plus anciens (1936) :

- la machine de Turing (Alan M. TURING)
- le λ-calcul (Alonzo CHURCH)
- les fonctions récursives (Stephen C. KLEENE)

Propriétés :

- ces modèles de calcul sont tous équivalents (i.e. réductibles les uns dans les autres)
- ces modèles permettent de construire tous les programmes élaborables par l'Homme

  Ces modèles de calcul sont dits ***universels***
- il existe des modèles de calcul plus faibles (i.e. certains problèmes solubles par une machine de Turing ne pourront jamais être résolus par de tels modèles).

  Exemple : machine de Turing à mémoire bornée (nombre fini de cases mémoire)

Terminologie

Les algorithmes

Un algorithme est une description destinée à un individu

$\Rightarrow$ en général, un mélange de français, de mathématiques et d'informatique !

Il définit les étapes à suivre pour résoudre un problème donné.

Exemple : étant donnés deux entiers n et m, comment calculer leur PGCD ? Voici un algorithme :

$$\begin{aligned} \text{PGCD}(n, m) &= n && \text{si } n=m \\ \text{PGCD}(n, m) &= \text{PGCD}(n, m-n) && \text{si } n<m \\ \text{PGCD}(n, m) &= \text{PGCD}(n-m, m) && \text{si } n>m \end{aligned}$$

Un algorithme doit :

- être exhaustif (tous les cas du problème à résoudre doivent pouvoir être envisagés)
- être non ambigu : à tout instant, le choix de la prochaine étape à réaliser est parfaitement défini
- s'arrêter au bout d'un nombre fini d'étapes

Accessoirement, un algorithme sera :

- le plus simple possible
- le plus efficace possible (i.e. le nombre d'étapes est le plus petit possible)

Les programmes

Dès lors qu'un algorithme est concrétisé effectivement dans un modèle de calcul (abstrait ou concret - au hasard, un ordinateur), il devient ***programme***.

Les fonctions

Quelques rappels

Soient deux fonctions $f : X \to Y$ et $g : Y \to Z$

- f est ***injective*** ssi $f(x) = f(y) \Leftrightarrow x = y$
- f est ***surjective*** ssi $\forall y \in Y, \exists x \in X \mid y = f(x)$
- f est ***bijective*** ssi f injective et surjective
- $h = g \circ f$ est une fonction $h : X \to Z$, dite ***composée*** de f et g, telle que $h(x) = g(f(x))$

  L'opérateur $\circ$ est dit ***opérateur de composition***

Arité et notation de l'application des fonctions

$\oplus : X \to Y$ est dite ***unaire*** (un paramètre).

L'application de $\oplus$ à x se note de trois façons :

écriture ***fonctionnelle*** : $\oplus(x)$

écriture ***préfixe*** : $\oplus\ x$

écriture ***postfixe*** (ou ***suffixe***) : $x\ \oplus$

$\oplus : X \times Y \to Z$ est dite ***binaire*** (deux paramètres).

L'application de $\oplus$ à x et y se note de deux façons :

écriture ***fonctionnelle*** : $\oplus(x, y)$

écriture ***infixe*** : $x \oplus y$

$\oplus : X_1 \times ... \times X_n \to Y$ est dite ***n-aire*** (n paramètres), avec n>2

L'application de $\oplus$ à x_1, ..., x_n ne se note que d'une seule façon :

écriture ***fonctionnelle*** : $\oplus(x_1, ..., x_n)$

Les fonctions

La notation λ

Contexte : la définition et l'application d'une fonction ne sont pas distinguables avec la notation classique :

exemple : si le contexte ne nous indique rien,

si $f(x) = 3x$ **alors** ...

doit-il être interprété comme :

- si la définition de f est $f(x) = 3x$ alors
- si le résultat de l'application de f sur x est $3x$ alors

La notation λ permet de lever cette ambiguïté :

- pour une définition de fonction,
  au lieu d'écrire : $f(x) = 3x$
  nous écrirons parfois : $f = \lambda x \,.\, 3x$
- pour une application de fonction,
  nous écrirons usuellement : $f(x)$

Avantage : évite de nommer explicitement une fonction

exemple : si d est une fonction qui, appliquée à f et x, calcule $d(f, x) = 2.f(x)$,

alors d appliquée à la fonction $\cos(x)+3$ et $\sin(y)$ peut s'écrire $d(\lambda x \,.\, \cos(x)+3, \sin(y))$

terme gauche : il s'agit d'une définition anonyme (sans nommage explicite) de fonction

terme droit : il s'agit d'une banale expression de calcul

Ecrire des algorithmes

cadre d'expression

Il existe essentiellement trois cadres d'expression des algorithmes :

- le cadre fonctionnel

  $\Rightarrow$ les langages de programmation correspondants sont dits ***langages de programmation fonctionnelle*** (LISP, APL, SCHEME, ML, CAML, HASKELL, GOFER, HUGS...)

- le cadre impératif

  $\Rightarrow$ ***langages de programmation impérative*** (C, PASCAL, BASIC, FORTRAN, ADA, C++, ...)

- le cadre logique (ou relationnel)

  $\Rightarrow$ ***langages de programmation logique*** (PROLOG, ...)

Nous travaillerons dans le cadre fonctionnel :

+ nécessite peu de notions de base
+ les techniques de preuve lui sont adaptées
+ l'analyse de la complexité est plus facile
+ mise en pratique très rapide
– la majorité des langages de programmation exploités dans le monde scientifique (hors informatique) sont impératifs : FORTRAN, ADA, PASCAL ou C.

L'algorithmique fonctionnelle

Généralités

Un problème est caractérisé par des variables e_1, ..., e_n liées par des contraintes.

Un algorithme solutionnant ce problème consiste à écrire une ***définition*** *f* de la forme :

$f(e_1, ..., e_n) = s_1$ **si** condition$_1$ sur les e_1, ..., e_n

...

$f(e_1, ..., e_n) = s_m$ **si** condition$_m$ sur les e_1, ..., e_n

sachant que :

- chaque ligne est appelée une ***clause***
- les *m* clauses couvrent tous les cas de figure des e_1, ..., e_n, et sont toutes exclusives (exhaustivité, non ambiguïté)
- chaque s_i est une expression de calcul quelconque, pouvant elle-même s'appuyer sur d'autres algorithmes.

Attention

le symbole = ne doit pas prêter à confusion dans la ligne :

$f(e) = s$ **si** $c(e)$

- il ne s'agit pas de l'égalité mathématique
- il faut plutôt l'interpréter comme "*se reformule en*" :

  $f(e) = s$ **si** $c(e)$ doit donc se lire ainsi :

  le problème (f appliqué à e) se reformule en le problème s si la condition c est vérifiée sur e

  Autrement dit, résoudre (f appliqué à e) revient à résoudre le problème s si $c(e)$ est vérifiée

L'algorithmique fonctionnelle

Spécification

Lors de l'écriture d'un algorithme, nous écrirons :

- la signification à donner aux variables e_1, ..., e_n
- les contraintes qu'elles doivent satisfaire pour que l'algorithme ait un sens
- les propriétés de la valeur calculée

Un exemple

Enoncé : écrire un algorithme capable de calculer le nombre de solutions de l'équation $a.x+b = 0$, pour a et b donnés.

Solution : l'algorithme NBSOL1 suivant répond au problème :

NBSOL1(a, b) :

entrée : a et b, deux réels quelconques

sortie : un entier naturel n, nombre de solutions de l'équation $a.x+b = 0$, avec les conventions suivantes :

n = 0	aucune solution
n = 1	une solution
n = -1	infinité de solutions

définition :

NBSOL1(a, b) = 1 **si** $a \neq 0$

NBSOL1(a, b) = 0 **si** $a = 0$ et $b \neq 0$

NBSOL1(a, b) = -1 **si** $a = 0$ et $b = 0$

L'algorithmique fonctionnelle

Récursivité

Un algorithme est dit ***récursif*** si, dans sa définition, il s'appuie sur lui-même.

Un exemple d'algorithme récursif

Enoncé : écrire un algorithme capable de calculer la somme de deux entiers naturels *a* et *b* n'utilisant que les opérations de calcul suivantes : ajouter ou soustraire 1.

Solution : l'algorithme récursif SUM suivant répond au problème :

SUM(a, b) :

entrée : *a* et *b*, deux entiers naturels

sortie : un entier naturel *s* tel que $s = a+b$

définition :

$\text{SUM}(a, b) = a$ **si** $b = 0$

$\text{SUM}(a, b) = \text{SUM}(a, b-1)+1$ **si** $b > 0$

Propriétés requises des définitions récursives

Dans une définition récursive :

- il doit exister au moins une clause qui n'est pas récursive (dite clause d'***arrêt***, ou de ***terminaison***)
- veiller à la convergence du calcul (i.e. toute clause de définition doit, lorsqu'elle est appliquée, aboutir à une clause d'arrêt.

Par exemple, la deuxième clause de SUM ne pourrait pas être remplacée par : $\text{SUM}(a, b) = \text{SUM}(a, b+1)-1$ **si** $b > 0$

L'algorithmique fonctionnelle

Récursivité terminale ou non terminale

Une définition récursive sera dite ***non terminale*** s'il existe au moins une clause telle que :

- la clause procède à un appel récursif
- l'élaboration du résultat nécessite de mémoriser le contexte du calcul, car l'appel récursif ne retourne pas le résultat final (il ne s'agit donc que d'un résultat intermédiaire)

Sinon la définition récursive est dite ***terminale***.

exemple : la définition de SUM est récursive non terminale :

SUM(a, b) = a **si** $b = 0$

SUM(a, b) = SUM(a, b-1)+1 **si** $b > 0$

Sur un calcul concret, cet algorithme produit :

SUM(4, 3) = SUM(4, 2)+1 = 7

SUM(4, 2) = SUM(4, 1)+1 = 6

SUM(4, 1) = SUM(4, 0)+1 = 5

SUM(4, 0) = 4

Cette autre définition de SUM est, par contre, récursive terminale :

SUM(a, b) = a **si** $b = 0$

SUM(a, b) = SUM(a+1, b-1) **si** $b > 0$

Sur un calcul concret :

SUM(4, 3) = SUM(5, 2)

SUM(5, 2) = SUM(6, 1)

SUM(6, 1) = SUM(7, 0)

SUM(7, 0) = 7

Un langage de programmation fonctionnelle : SCHEME

Généralités

SCHEME est un langage dérivé du langage de programmation LISP (début des années 60), comme le fut aussi LOGO.

SCHEME n'est pas un langage de programmation fonctionnelle pur :

⇒ une même expression calculée dans une même session de travail ne produit pas nécessairement le même résultat.

Exemple :

```
(define un 1)
(define (plus_un x) (+ x un))
(plus_un 2)
```

⇒ a pour valeur 3

même expression

```
(define un 10)
(plus_un 2)
```

⇒ a pour valeur 12

Dans ce cours

Le langage que vous apprendrez est SCHEME.

Toutes les corrections d'exercices seront données dans trois langages différents : SCHEME, CAML et HASKELL

⇒ l'objectif est de montrer que :

- le but de ce cours n'est pas l'apprentissage d'un langage de programmation particulier, mais apprendre à concevoir des algorithmes dans un cadre fonctionnel.
- chaque langage de programmation a son propre système d'écriture des programmes, mais tous partagent les mêmes principes (ici fonctionnels) de conception des programmes.

Principes de base

L'interpréteur SCHEME fonctionne ainsi :

partant de l'***environnement de calcul*** courant :

1. attendre que l'usager saisisse une ***expression*** à calculer
2. ***évaluer*** l'expression saisie, i.e. calculer sa ***valeur***
3. ***afficher*** la valeur à l'usager

L'environnement de calcul contient toutes les définitions (appelées ***liaisons***) connues de l'interpréteur.

L'évaluation d'une expression (point 2) peut transformer l'environnement de calcul courant.

La prochaine expression sera évaluée dans ce nouvel environnement.

Exemple (en **gras**, ce que l'usager saisit) :

```
? 1
value: 1
? (+ 3 4)
value: 7
? un
error: unbound name "un"
? (define un 1)
value: un
? un
value: 1
? (+ un 4) ; un commentaire
value: 5
? (+
      un 8
    )
value: 9
```

Remarque : tout caractère situé après un ";" est ignoré jusqu'à la fin de la ligne ⇒ écriture d'un ***commentaire***

Expressions de base

Généralités

Dans ce qui suit, sont indiqués pour les expressions de calcul de base :

- comment elles doivent être écrites (niveau ***syntaxique***)
- le mécanisme d'évaluation associé (niveau ***sémantique***)

Les nombres

La plupart des langages distinguent les nombres entiers des nombres réels (codages en mémoire différents).

Les nombres entiers

syntaxe :

une séquence de chiffres tous collés les uns aux autres, éventuellement précédée d'un signe + ou -

exemple :

```
12   +459    -452
```

sémantique :

la valeur d'un nombre entier est cet entier lui-même.

exemple :

```
? 12
value: 12
? +23
value: 23
? -65
value: -65
```

Expressions de base

Les nombres flottants

Un nombre flottant est une représentation exacte ou approchée d'un nombre réel.

syntaxe :

écrire un nombre $x = m \times 10^e$, c'est écrire sa ***mantisse***, suivie ou non d'un ***exposant***.

- la mantisse est de la forme :

  partie_entière **.** partie_fractionnaire

- la ***partie_entière*** est un entier signé ou non
- la ***partie_fractionnaire*** est un entier non signé
- l'exposant commence par la lettre `e` ou `E`, suivie d'un entier signé ou non

exemple :

```
1.0    -3.12     1.0e+20     314.159e-2
```

sémantique :

la valeur d'un nombre flottant est ce flottant lui-même.

exemple :

```
? 2.05
value: 2.05
? -1.0e-2
value: -0.01
? +10e+3
value: 10000.
? 2e2
value: 200.
```

Remarque : la présence du caractère "." ou d'un exposant suffit à distinguer un entier d'un nombre flottant.

Expressions de base

Les noms (ou variables)

syntaxe :

un ***nom*** est une séquence de caractères (pas de distinction majuscules/minuscules) autres que :

```
( ) " ; ' ` [ ] | { }
```

et qui ne s'interprète pas comme un nombre

exemple :

```
un    vrai?    +    non!    Petit_nom
```

sémantique :

la valeur d'un nom est celle qui lui a été associée (nous dirons ***liée***) dans l'environnement courant. Si aucune valeur n'est liée au nom, l'interpréteur s'arrête et affiche un message d'erreur.

exemple :

```
? sin
value: <predefined function>
? +
value: <predefined function>
? un_nom
error: unbound name "un_nom"
```

Les booléens

Il existe deux noms prédéfinis ayant une signification particulière :

`#t` est le nom de la valeur de vérité vrai (*true*)

`#f` est le nom de la valeur de vérité faux (*false*)

Expressions de base

Les formes, ou expressions parenthésées

Dans le cas général, une ***forme*** permet de représenter une application de fonction à une liste de paramètres.

syntaxe :

une forme est une expression parenthésée du type :

$$(\ exp_0\ exp_1\ ...\ exp_n)$$

où chaque exp_i est une expression.

exemple :

```
(+ 3 45)
(* (+ (- 5 7) 7 8) (/ 3 4))
(sin (* pi 2))
```

Il existe une quarantaine de ***formes spéciales*** :

- les formes `begin`, `if`, `cond`, `lambda` et `define`
- et d'autres formes que nous n'étudierons pas, ou plus tard

Ces formes spéciales ont une sémantique particulière, que nous étudirons plus loin.

Les formes non spéciales sont dites ***formes communes***.

Sémantique des formes communes (non spéciales) :

1. évaluation de chacune des expressions exp_i, dans l'ordre d'apparition (de gauche à droite) ; soient v_i les valeurs calculées.
2. Si v_0 (valeur de exp_0) est une fonction prédéfinie, appliquer cette fonction sur les valeurs v_1 ... v_n , et retourner la valeur résultant de cette application comme celle de la forme.
3. Si v_0 est une valeur fonctionnelle, voir la forme spéciale `lambda`.
4. Sinon v_0 n'est pas du genre fonction ; arrêter alors le calcul en signalant une erreur.

exemple :

```
(+ 3 45)
```

a pour valeur 48 ; en effet, chaque expression est tout d'abord évaluée :

expression	**valeur**
`+`	fonction prédéfinie
`3`	3
`45`	45

La fonction prédéfinie + est alors appliquée à tous ses arguments, d'où le résultat.

exemple :

```
(* (+ (- 5 7) 7 8) (/ 8 4))
```

a pour valeur 26 ; en effet :

1. chaque expression est tout d'abord évaluée :

expression	**valeur**
`*`	fonction prédéfinie
`(+ (- 5 7) 7 8)`	13
`(/ 8 4)`	2

2. la fonction prédéfinie `*` est alors appliquée à tous ses arguments, d'où le résultat.

Notons que nous avons appliqué le principe d'évaluation sur les sous-expressions, par exemple `(+ (- 5 7) 7 8)`:

expression	**valeur**
`+`	fonction prédéfinie
`(- 5 7)`	-2
`7`	7
`8`	8

Quelques fonctions prédéfinies

forme	sémantique
`(+` e_1 ... e_n`)`	calcule la somme de tous les e_i
`(*` e_1 ... e_n`)`	calcule le produit de tous les e_i
`(-` e_1 ... e_n`)`	calcule e_1 `-` `(+` e_2 ... e_n`)`
`(/` e_1 e_2`)`	calcule la division de e_1 par e_2
`(and` e_1 ... e_n`)`	calcule le *et logique* des e_i
`(or` e_1 ... e_n`)`	calcule le *ou logique* des e_i
`(not` e_1`)`	calcule la *négation logique* de e_1
`(=` e_1 ... e_n`)`	retourne `#t` si $e_1 = ... = e_n$, `#f` sinon
`(>` e_1 ... e_n`)`	retourne `#t` si $e_1 > ... > e_n$, `#f` sinon
`(<` e_1 ... e_n`)`	retourne `#t` si $e_1 < ... < e_n$, `#f` sinon
`(>=` e_1 ... e_n`)`	retourne `#t` si $e_1 \geq ... \geq e_n$ e_2, `#f` sinon
`(<=` e_1 ... e_n`)`	retourne `#t` si $e_1 \leq ... \leq e_n$, `#f` sinon
`(max` e_1 ... e_n`)`	calcule le maximum de tous les e_i
`(min` e_1 ... e_n`)`	calcule le minimum de tous les e_i

`and` et `or` sont, en fait, des formes spéciales.

exemple :

forme	valeur
`(+ 1 2 3)`	6
`(/ 6 2)`	3
`(/ 7 2)`	3.5
`(= (+ 1 2) (+ 2 1))`	`#t`
`(not (= 3 4))`	`#t`
`(and (= 2 3) (= 4 5))`	`#f`

Les formes spéciales

Forme spéciale `begin` *ou* `sequence`

Cette forme permet d'évaluer une séquence d'expressions :

syntaxe :

`(sequence` exp_1 ... exp_n`)`

ou

`(begin` exp_1 ... exp_n`)`

où chaque exp_i est une expression.

exemple :

```
(begin 1 (+ 1 3) (* 2 4))
```

sémantique :

Chaque exp_i est évaluée, dans l'ordre de la séquence.

Si exp_i modifie l'environnement courant, exp_{i+1} est évaluée dans ce nouvel environnement.

La valeur de la forme est la valeur de la dernière expression.

exemple :

```
? (begin 1 (+ 1 3) (* 2 4))
value: 8
```

L'interpréteur a commencé par :

1. évaluer `1`, pour trouver 1
2. évaluer `(+ 1 3)`, pour trouver 4
3. évaluer `(* 2 4)`, pour trouver 8

La valeur retournée est la dernière calculée, soit 8.

Cette forme est utilisée lorsque le programmeur veut imposer un ordre d'évaluation des exp_i ; en effet, Scheme ne révèle rien sur l'ordre d'évaluation des autres formes.

Les formes spéciales

Forme spéciale `if`

Cette forme permet d'évaluer une expression au choix :

syntaxe :

`(if` exp_1 exp_2 exp_3`)`

où chaque exp_i est une expression.

exemple :

```
(if (= 0 (- 1 1)) 2 (+ 1 3))
```

sémantique :

1. Calcul de exp_1 ; soit v la valeur calculée.
2. Si v est égale à `#t` (vrai), alors exp_2 est évaluée, et sa valeur est celle de la forme
3. Si v est égale à `#f` (faux), alors exp_3 est évaluée, et sa valeur est celle de la forme

exemple :

```
? (if (= 0 (- 1 1)) 2 (+ 1 3))
value: 2
? (if (= 0 0) 1 (/ 1 0))
value: 1
? (if (> 0 0) 1 (/ 1 0))
error: division by 0
? (if (< 0 0) 1 (+ 3 4))
value: 7
```

Les formes spéciales

Forme spéciale `cond`

Cette forme est une extension de la forme spéciale `if` :

syntaxe :

(cond ($test_1$ exp_{11} ... exp_{1n})
($test_2$ exp_{21} ... exp_{2n})
...
($test_m$ exp_{m1} ... exp_{mn})
(else exp_1 ... exp_n))

où chaque exp_{ij} est une expression, et $test_i$ une expression ayant un résultat booléen.

exemple :

```
(cond ((= x 0) 1)
      ((= x 1) 2)
      ((= x 4) 3)
      (else    4))
```

sémantique :

L'expression est équivalente à :

(if $test_1$ (sequence exp_{11} ... exp_{1n})
(if $test_2$ (sequence exp_{21} ... exp_{2n})
...
(if $test_m$ (sequence exp_{m1} ... exp_{mn})
(sequence exp_1 ... exp_n))...)

exemple :

```
(cond ((= x 0) 1)
      ((= x 1) 2)
      ((= x 4) 3)
      (else    4))
```

si *x* vaut 1, la valeur est 2 ; si *x* vaut 3, la valeur est 4.

Les formes spéciales

Forme spéciale `define`

syntaxe :

(`define` *nom* exp_1 ... exp_n)

ou

(`define` (*nom* p_1 ... p_m) exp_1 ... exp_n)

où chaque exp_i est une expression, chaque p_i et *nom* des noms.

exemple :

```
(define un 1)
(define (plus_un x) (+ 1 x))
(define (distance1D a b)(abs (- a b)))
```

sémantique :

a) dans l'environnement courant,

(`define` *nom* exp_1 ... exp_n)

lie le *nom* à la valeur *v* de l'expression :

(`begin` exp_1 ... exp_n)

Si ce nom était déjà lié à une valeur dans l'environnement, la valeur *v* remplace cette ancienne valeur.

Sinon une nouvelle liaison (*nom*, *v*) est créée.

b) dans l'environnement courant,

(`define` (*nom* p_1 ... p_m) exp_1 ... exp_n)

lie le *nom* à la valeur de l'expression :

(`lambda` (p_1 ... p_m)(`begin` exp_1 ... exp_n))

Autrement dit, la seconde forme est équivalente à :

(define *nom*
        (lambda (*p*$_1$ ... *p*$_m$)
                (begin *exp*$_1$ ... *exp*$_n$)))

Remarque : équivalence correcte si la fonction définie n'est pas récursive ; sinon, c'est une `named-lambda` (une variante de la forme spéciale `lambda`)

exemple : soit *E* l'environnement courant :

```
? (define pi 3.1415927)
value: pi
```

E a été transformé en E_1, où le nom `pi` est maintenant lié à la valeur 3,1415927.

```
? (define (circonférence r) (* 2 pi r))
value: circonférence
```

E_1 a été transformé en E_2, où le nom `circonférence` est maintenant lié à une valeur fonctionnelle.

```
? pi
value: 3.1415927
```

E_2 n'est pas modifié ; la valeur associée au nom `pi` est bien celle annoncée.

```
? (define pi 4)
value: pi
```

E_2 a été transformé en E_3, où le nom `pi` est maintenant lié à la valeur 4 ; l'ancienne valeur est perdue.

```
? pi
value: 4
```

E_3 n'est pas modifié ; la valeur associée au nom `pi` est la dernière lui ayant été liée, soit 4.

```
? (circonférence 1)
value: 8
```

E_3 n'est pas modifié, mais la fonction `circonférence` ne produit plus le résultat escompté !

Les formes spéciales

Forme spéciale `lambda`

Cette forme permet de construire des fonctions, dites ***anonymes*** (i.e. sans nom) :

syntaxe :

`(lambda (`p_1 ... p_m`)` exp_1 ... exp_n`)`

où chaque exp_i est une expression, chaque p_i un nom.

exemple :

```
(lambda (x) (+ 1 x))
(lambda (a b)(abs (- a b)))
```

sémantique :

la valeur d'une telle forme est une ***valeur fonctionnelle***, caractérisée par un triplet :

- l'état de l'environnement courant (i.e. présent au moment de l'évaluation de la forme `lambda`), appelé aussi ***fermeture***
- les noms de ses ***paramètres*** p_1 ...p_m
- son ***corps***, séquence des exp_i

exemple :

```
? (lambda (x) (+ x x))
value: <user function>
```

sémantique de l'application de fonction `lambda`

Si une forme commune du genre :

(exp_0 exp_1 ... exp_n)

est évaluée,

et que la valeur de exp_0 est une valeur fonctionnelle v_0 (donc résultat d'une forme `lambda`), soit un triplet :

- F : fermeture (environnement de définition de la `lambda`)
- p_1 ... p_n : paramètres
- c : corps

alors, si chaque exp_i a été évaluée à une valeur v_i (soit E l'environnement après toutes ces évaluations), la valeur finale de la forme commune est celle de :

```
(begin
   (define p1 v1)
   (define p2 v2)
   ...
   (define pn vn)
   c)
```

évaluée dans l'environnement F.

Après évaluation, toutes les liaisons p_i créées par les `define` de sont défaites, et l'environnement final redevient E.

exemple : soit à évaluer ces expressions, l'environnement initial étant E :

```
(define un 1)
(define (plus_un x) (+ un x))
(plus_un 3)
(define x 2)
(define (f y) (+ (plus_un (* 2 x)) x y))
(f 3)
x
```

a) évaluation du premier `define` dans l'environnement E :

```
(define un 1)
```

Par définition de cette forme spéciale `define`, évaluons l'expression `1`, qui a pour valeur 1.

Modifions l'environnement courant E en un environnement E_1 identique à E, sauf pour le nom `un` lié à 1.

Le résultat est le nom de la liaison créée, soit `un`.

b) évaluation du second `define` dans l'environnement hérité de l'étape précédente, donc E_1 :

```
(define (plus_un x) (+ un x))
```

Par définition de `define`, nous devons en fait évaluer :

```
(define plus_un (lambda (x) (+ un x)))
```

Evaluons l'expression `(lambda (x) (+ un x))`, qui s'évalue en une valeur fonctionnelle v_1 caractérisée par :

- sa fermeture E_1 (environnement au moment de l'évaluation)
- son paramètre `x`
- son corps `(+ un x)`

Reste à modifier l'environnement courant E_1 en un environnement E_2 en tous points identique à E_1, sauf pour le nom `plus_un`, qu'il lie à la valeur fonctionnelle v_1.

Le résultat est le nom de la liaison créée, soit `un`.

c) évaluation de la forme commune dans l'environnement hérité de l'étape précédente, donc E_2 :

```
(plus_un 3)
```

Commençons par évaluer chacune des expressions composant cette forme :

`plus_un` ⇒ valeur fonctionnelle v_1

`3` ⇒ entier 3

La première expression de la forme a une valeur fonctionnelle ; nous savons alors que, par définition, la valeur de notre forme originale est la valeur de :

```
(begin
   (define x 3)
   (+ un x))
```

dans l'environnement E_1 (fermeture de v_1) ;

c.1) `(define x 3)` transforme E_1 en E_3, en liant `x` à 3

c.2) `(+ un x)` est une forme commune :

`+` ⇒ fonction prédéfinie

`un` ⇒ entier 1

`x` ⇒ entier 3

La valeur calculée est donc 4.

C'est aussi la valeur de `begin` (dernière expression).

L'application de la valeur fonctionnelle produit donc le résultat 4.

L'évaluation de l'application de la valeur fonctionnelle étant achevée, l'environnement est restauré dans son état d'avant cette évaluation, soit l'environnement E_2.

d) évaluation du troisième `define` dans l'environnement E_2 :

```
(define x 2)
```

Modifions l'environnement courant E_2 en un environnement E_4 identique à E_2, sauf pour le nom `x`, lié à la valeur 2.

Le résultat est le nom de la liaison créée, soit `x`.

e) évaluation du dernier `define` dans l'environnement hérité de l'étape précédente, donc E_4 :

```
(define (f y) (+ (plus_un (* 2 x)) x y))
```

Par définition de `define`, nous devons en fait évaluer :

```
(define f
        (lambda (y)
             (+ (plus_un (* 2 x)) x y)))
```

Evaluons l'expression, qui s'évalue en une valeur fonctionnelle v_2 caractérisée par :

- sa fermeture E_4 (environnement au moment de l'évaluation)
- son paramètre `y`
- son corps `(+ (plus_un (* 2 x)) x y)`

Il nous reste à modifier l'environnement courant E_4 en un environnement E_5 identique à E_4, sauf pour le nom `f`, lié à la valeur fonctionnelle v_2.

Le résultat est le nom de la liaison créée, soit `f`.

f) évaluation de la forme commune dans l'environnement hérité de l'étape précédente, donc E_2 :

```
(f 3)
```

Commençons par évaluer chacune des expressions composant cette forme :

`f` $\Rightarrow$ valeur fonctionnelle v_2

`3` $\Rightarrow$ entier 3

La première expression de la forme a une valeur fonctionnelle ; nous savons alors que, par définition, la valeur de notre forme originale est la valeur de :

```
(begin
   (define y 3)
   (+ (plus_un (* 2 x)) x y))
```

dans l'environnement E_4 (fermeture de v_2) ;

f.1) `(define y 3)` transforme E_4 en E_5, en liant `y` à 3

f.2) `(+ (plus_un (* 2 x)) x y)` est une forme commune :

`+`	$\Rightarrow$ fonction prédéfinie
`(plus_un (* 2 x))`	$\Rightarrow$ forme commune à évaluer
`x`	$\Rightarrow$ 2
`y`	$\Rightarrow$ 3

f.2.1) calcul de `(plus_un (* 2 x))` dans E_5 :

`plus_un`	$\Rightarrow$ valeur fonctionnelle v_1
`(* 2 x)`	$\Rightarrow$ 4, puis `x` vaut 2 dans E_5

Puisqu'il s'agit d'appliquer une valeur fonctionnelle, nous devons calculer :

```
(begin
   (define x 4)
   (+ un x))
```

dans l'environnement E_1 (fermeture de v_1).

La valeur calculée est 5 (ajout de 1 à 4). Le calcul achevé, le `define` du `begin` est défait, i.e. l'environnement retrouve son état d'avant le `begin`, donc redevient E_5.

Nous pouvons donc en déduire la valeur de la forme en f.2) : il s'agit de 10.

Le calcul de `(f 3)` est achevé : l'environnement retrouve donc son état d'avant le `begin`, soit E_2.

g) évaluation de `x` dans l'environnement hérité du calcul précédent, soit E_2.

La valeur est 2.

Les formes spéciales

Formes spéciales `let` *et* `let*`

Ces deux formes sont en fait des variantes syntaxiques de la forme spéciale `lambda` :

syntaxe :

(let ((*n*1 *v*1) ... (*n*k *v*k)) *e*1 ... *e*m)

(let* ((*n*1 ... *n*k) *v*1 ... *v*k) *e*1 ... *e*m)

où chaque e_i ou v_i est une expression, chaque n_i un nom.

exemple :

```
(let ((x 1)(y 2)) (+ x y 2))
(let ((a (+ 4 5)))(- a))
```

sémantique :

la valeur de ces deux formes est strictement équivalente à celle de l'application d'une fonction anonyme suivante :

((lambda (*n*1 ... *n*k) *e*1 ... *e*m) *v*1 ... *v*k)

exemple :

```
? (let ((x 2)(y 4))
        (+ (let ((x 1)) (+ x y))
           x))
value: 7
```

Les chaînes de caractères

Les littéraux

La syntaxe d'un littéral ***chaîne de caractères*** (ou ***texte***) est de la forme :

une séquence de caractères encadrée de guillemets (")

Le caractère anti-slash (\) est spécial : il précède tout caractère dit ***spécial***, comme par exemple " ou \.

Exemples : trois textes

```
"un texte"
"un guillemet (\") est un caractère"
"l'anti-slash (\\) aussi"
```

Quelques fonctions prédéfinies

`(string?` *v*`)`	retourne `#t` si *v* est un texte, `#f` sinon
`(string-null?` *v*`)`	retourne `#t` si *v* est un texte vide (""), `#f` sinon
`(string-length` *v*`)`	retourne le nombre de caractères de *v* (longueur du texte)
`(string=?` *v w*`)`	retourne #t si *v* et *w* sont 2 textes identiques, `#f` sinon
`(substring` *t d f*`)`	retourne un texte composé de tous les caractères de *t* dont la position est comprise entre *d* (inclus) et *f* (exclu) Le premier caractère d'un texte est en position 0, le dernier en position *n*-1 (où *n* est la longueur du texte)

Les chaînes de caractères

Quelques exemples

Supposons que soient définies les variables suivantes :

```
(define x "Bonjour")
(define y "bonjour")
(define z 1)
```

alors :

expression	valeur
`(string? x)`	`#t`
`(string? z)`	`#f`
`(string-null? x)`	`#f`
`(string-null? "")`	`#t`
`(string-length x)`	`7`
`(string=? x x)`	`#t`
`(string=? x y)`	`#f`
`(substring x 0 3)`	`"Bon"`
`(substring y 3 7)`	`"jour"`

Les paires

Définition

Une ***paire*** est composée d'un couple de valeurs :

- la première composante est appelée le ***car***
- la seconde est appelée le ***cdr***

Fonctions prédéfinies

Si v_1 et v_2 sont deux valeurs, alors

`(cons` v_1 v_2`)`

retourne une paire dont le car est v_1, le cdr est v_2

Une telle paire est représentée ainsi (notation pointée) :

`(`v_1 `.` v_2`)`

Si *p* est une paire, alors :

`(car` *p*`)` retourne le car de la paire

`(cdr` *p*`)` retourne le cdr de la paire

Si *v* est une valeur quelconque (paire ou non), alors :

`(pair? v)` retourne `#t` si *v* est une paire, `#f` sinon

Abréviations

`(caar` *p*`) = (car (car` *p*`))`

`(cadr` *p*`) = (car (cdr` *p*`))`

`(cdar` *p*`) = (cdr (car` *p*`))`

`(cddr` *p*`) = (cdr (cdr` *p*`))`

`(caaar` *p*`) = (car (car (car` *p*`)))`

... (toutes les combinaisons à 3 et 4)

`(cddddr` *p*`) =(cdr (cdr (cdr (cdr` *p*`))))`

Les listes

Définition

Une liste est soit :

- la liste vide `()`
- une paire `(`*e* `.` *l*`)`, où *l* est une liste

exemples :

`(1 . (2 . (3 . ())))` est une liste

`(1 . (2 . 3))` n'est pas une liste

Notation

Toute liste de la forme :

`(`e_1 `.` `(`e_2 `.` `(` ... `.` `(`e_n `.` `())` ... `)))`

sera notée plus simplement :

`(`e_1 e_2 ... e_n`)`

exemple :

`(1 . (2 . (3 . ())))` sera notée `(1 2 3)`

Fonctions prédéfinies

`(list` exp_1 `...` exp_n`)`	retourne la liste `(`v_1 v_2 `...` v_n`)`, où chaque v_i est la valeur de exp_i
`(list?` *v*`)`	retourne `#t` si *v* est une liste, `#f` sinon
`(null?` *v*`)`	retourne `#t` si *v* est la liste vide `()`, `#f` sinon

Les listes

Quelques exemples

expression	valeur
`(list 1 (+ 1 1) (* 3 1))`	`(1 2 3)`
`(list? ())`	`#t`
`(list? (cons 1 ()))`	`#t`
`(list? 1)`	`#f`
`(null? ())`	`#t`
`(null? (cons 1 ()))`	`#f`

Les noms sont aussi des valeurs

La forme spéciale `quote`

Cette forme permet de ne pas évaluer une expression :

syntaxe :

`(quote` *e*`)`

ou

`'`*e*

exemple :

```
(quote (+ 1 3))
```

sémantique :

Retourne l'expression *e* telle quelle, sans évaluation.

exemple :

```
? (quote (+ 1 3))
value: (+ 1 3)
? '(+ 1 3)
value: (+ 1 3)
? (list '+ 1 3)
value: (+ 1 3)
? (list '(+ 1 3) (+ 1 3))
value: ((+ 1 3) 4)
? ''quote
value: 'quote
```

Les noms sont aussi des valeurs

La fonction prédéfinie `eval`

Cette fonction procède à une double évaluation de son argument.

syntaxe :

`(eval` *e*`)` où *e* est une expression

sémantique :

`eval` étant une fonction prédéfinie, l'interpréteur évalue normalement *e* une première fois, pour obtenir une valeur *v*.

Cette valeur *v* est ensuite réévaluée, afin de produire le résultat final.

exemple :

```
? (define x (list '+ (+ 2 3) '(+ 2 3)))
value: x
? x
value: (+ 5 (+ 2 3))
? (eval x)
value: 10
```

Les types abstraits

Définition d'un type

En informatique, à un ***type de donnée*** (ou plus simplement ***type***) sont associés :

- un nom
- un ensemble particulier de valeurs
- un ensemble de fonctions permettant de manipuler ces valeurs

Le codage en mémoire des valeurs est spécifique du type.

exemple : nous avons étudié un certain nombre de types prédéfinis en SCHEME : les entiers, les réels, les chaînes de caractères, les paires, les listes

Pourquoi type abstrait

La notion d'abstraction indique que les manipulations faites sur les valeurs du type sont indépendantes du codage de ces valeurs.

Constructeurs et sélecteurs

un ***constructeur*** d'un type T est une fonction dont le résultat est une valeur de type T.

Toutes les valeurs d'un type T peuvent être obtenues par combinaison de constructeurs de T.

un ***sélecteur*** d'un type T est une fonction qui s'applique à une valeur d'un type T, et qui retourne une valeur qui n'est pas de type T.

Les sélecteurs doivent permettre de retrouver comment une valeur a été construite

Les types abstraits

Quelques exemples

Pour les entiers naturels, les constructeurs sont la constante 0 et la fonction successeur ***succ*** (succ(x) = x+1)

En effet, n'importe quel entier peut être exprimé comme une combinaison de ces constructeurs

exemple : 3 = succ(succ(succ(0)))

Pour les listes, les constructeurs sont la liste vide et la fonction ***cons*** ; ses sélecteurs sont les fonctions ***car*** et ***cdr***.

Un exemple : les ensembles d'entiers

Si nous devions définir un nouveau type dans SCHEME permettant de manipuler des ensembles quelconques composés d'entiers, nous devrions indiquer :

- les constructeurs du type

Par exemple, nous pourrions choisir :

`vide` la constante représentant l'ensemble vide

`(ajouter` *e S*`)` la fonction qui retourne l'ensemble $S \cup \{e\}$, où *S* est un ensemble, et *e* un entier

Ces 2 constructeurs permettent bien d'engendrer n'importe quel ensemble d'entiers.

A ces constructeurs seraient associés deux prédicats, le premier nous permettant de tester si une valeur est un ensemble, le second nous permettant de retrouver le dernier constructeur utilisé pour construire l'ensemble :

`(ensemble?` *v*`)` retourne `#t` si *v* est un ensemble, `#f` sinon

`(vide?` *E*`)` retourne `#t` si *E* est l'ensemble vide, `#f` sinon

- les sélecteurs

Nous pourrions choisir ici :

`(premier` *E*`)` retourne le plus petit entier élément d'un ensemble *E* non vide

`(autres` *E*`)` retourne l'ensemble *E* (supposé non vide) privé de son plus petit élément

Ces 2 sélecteurs permettent bien de décomposer n'importe quel ensemble non vide, afin de retrouver comment il a été construit

- les fonctions de manipulation

il s'agirait simplement de définir les fonctions usuelles sur les ensembles :

`(appartient?` e S`)` retourne `#t` si $e \in S$, `#f` sinon

`(inclus?` A B`)` retourne `#t` si $A \subseteq B$, `#f` sinon

`(ens+` A B`)` retourne l'ensemble $A \cup B$

`(ens*` A B`)` retourne l'ensemble $A \cap B$

`(ens-` A B`)` retourne l'ensemble A-B

... etc

Tout ***utilisateur*** de ce type abstrait manipule les ensembles via les fonctions définies ci-dessus.

Tout ***concepteur*** de ce type abstrait doit, par contre, définir toutes les fonctions ci-dessus, en s'appuyant sur des types déjà existants. Il n'a pas nécessairement qu'une seule façon de procéder.

Table des matières