Programmation fonctionnelle en langage Scheme

- Support PG104 -

Myriam Desainte-Catherine et David Renault

ENSEIRB-MATMECA, département d'informatique

January 21, 2020

Objectifs pédagogiques

Objectif général

Découverte de la programmation fonctionnelle pure à travers :

- Les formes qu'elle prend syntaxiquement (expressions, fonctions et listes récursives)
- Les méthodes qu'elle permet de déployer.

Compétences générales attendues

- Spécifier un calcul de façon fonctionnelle plutôt qu'impérative : programmer au moyen d'expressions plutôt que d'instructions
- Spécifier des calculs récursivement plutôt qu'itérativement
- Spécifier un calcul générique : abstraire un calcul au moyen de paramètres fonctionnels et de retours fonctionnels
- Comparer des solutions selon le style et la complexité

Ce support est accessible en version électronique mise à jour régulièrement aux adresses suivantes :
http://www.labri.fr/perso/myriam/Enseignement/Scheme/scheme.pdf
http://www.labri.fr/perso/renault/working/teaching/schemeprog/schemeprog.php

Documentation : https://docs.racket-lang.org/

Concepts et terminologie

Concepts fonctionnels

- **Écriture fonctionnelle** : programmation par applications de fonctions plutôt que par l'exécution de séquences d'instructions
- **Transparence référentielle** : chaque expression peut être remplacée par son résultat sans changer le comportement du programme – sans effets de bord[a]
- **Programmation fonctionnelle pure** : sans effets de bords, avec transparence référentielle.
- **Fonctions de première classe** : type fonction, constantes fonction, opérateurs sur les fonctions

[a] Un effet de bord est une modification de l'environnement (affectation ou E/S)

Autres concepts nouveaux

- **Typage dynamique** : Les variables sont typées au moment de l'exécution et non au moment de la compilation
- **Références** : ce sont des adresses sur des objets, elles sont utilisées chaque fois que les contenus ne sont pas utiles (passages de paramètres, retours de fonctions)
- **Garbage collector ou ramasse-miettes** : gestion dynamique et automatique de la mémoire. L'utilisateur ne s'occupe pas de désallouer la mémoire.

Transparence référentielle en C

Une fonction qui ne renvoie pas le même résultat pour une configuration d'arguments donnée et/ou qui modifie l'environnement ne respecte pas la transparence référentielle.

```
int y = 0;
int f() {
  return y++;
}
```

```
void g() {
    int i= f();  /* renvoie 1 */
    int j= f();  /* renvoie 2 */
  }
```

```
int y = 0;
int f(int x) {
  y = y+1;
  return x;
}
```

```
void g() {
    int i= f(1);  /* i=1 -- y=1 */
    int j= f(1);  /* j=1 -- y=2 */
  }
```

Ces fonctions de la bibliothèque C respectent-elles la transparence référentielle?

- int abs(int j); oui ou non
- int rand(void); oui ou non

Historique

Les langages des années 1950

- FORTRAN (1954) : calcul scientifique, données et calcul numérique.
- Lisp (1958) : calcul symbolique, données et algorithmes complexes (IA), démonstrations automatiques, jeux etc.
- Algol (1958) : langage algorithmique et structuré, récursivité.

Les langages lisp

- 1958 : John Mac Carthy (MIT)
- 1986 : common lisp ANSI (portabilité, consistence, expressivité, efficacité, stabilité).
- Les enfants de lisp :
  - Logo (1968), langage visuel pédagogique
  - Smalltalk (1969, Palo Alto Research Center de Xerox) premier langage orienté objets
  - ML (1973, R. Milner, University of Edinburgh), preuves formelles, typage statique, puis CaML (1987 INRIA), projet coq, et Haskell purement fonctionnel, paresseux.
  - Scheme (1975, Steele et Sussman MIT) mieux défini sémantiquement, portée lexicale, fermeture, et continuations de première classe.
  - emacs-lisp (Stallman 1975, Gosling 1982), langage d'extension de GNU Emacs.
  - CLOS (1989, Common Lisp Object System), common lisp orienté objets.

Le λ-calcul

Théorie des fonctions d'Alonzo Church (1930), modèle universel de calcul, directeur de thèse d'Alan Turing (machines de Turing, théorie de la calculabilité).

Syntaxe – λ-termes

- **Variables** : x, y, ...
- **Applications** : si u et v sont des λ-termes uv est aussi un λ-terme. On peut alors voir u comme une fonction et v comme un argument, uv étant alors l'image de v par la fonction u.
- **Abstractions** : si x est une variable et u un λ-terme alors $\lambda x.u$ est un λ-terme. Intuitivement, $\lambda x.u$ est la fonction qui à x associe u.

Exemple

- Constante : $\lambda x.y$
- Identité : $\lambda x.x$
- Fonction renvoyant une fonction : $\lambda x.\lambda y.a$
- Application : xyz ou $((xy)z)$
- Fonctions à plusieurs arguments : $\lambda xy.a$

Remarques: les applications sont faites de gauche à droite en l'absence de parenthèses, une occurrence de variable est dite **muette** ou **liée** si elle apparaît dans le corps d'un λ-terme dont elle est paramètre, sinon elle est dite **libre**.

Syntaxe du λ-calcul

Soient x et y des variables et u et v des λ-termes

- Les λ-termes suivants sont-ils bien formés : oui ou non
  - x
  - λx . u
  - $\lambda\lambda$x. u
  - λx u
  - (λx . u) y
  - (λu . λu) y
  - ((x u) y)
- Dans les λ-termes suivants, les occurences de la variable `x` sont-elles libres ou liées ?
  - ((u x) y)
  - λy . u x
  - λx . u x

Le λ-calcul – la substitution

Cette opération permet de remplacer les occurrences d'une variable par un terme pour réaliser le calcul des λ-termes. On note **t[x := u]** la substitution dans un lambda terme t de toutes les occurrences d'une variable x par un terme u.

Exemple

Dans ces exemples, les symboles x, y, z, a sont des variables.

- Dans une application : $xyz[y := a] = xaz$
- Dans une abstraction (cas normal) : $\lambda x.xy[y := a] = \lambda x.xa$
- Capture de variable libre : $\lambda x.xy[y := ax] = \lambda z.zax$(et non $\lambda x.xax$), renommage de la variable liée
- Substitution inopérante (sur variable liée): $\lambda x.xy[x := z] = \lambda z.zy = \lambda x.xy$

Définition

- **Variable**: si t est une variable alors $t[x := u] = u$ si $x = t$ et t sinon
- **Application**: si $t = vw$ alors $t[x := u] = v[x := u]w[x := u]$ si v et w sont des termes.
- **Abstraction**: si $t = \lambda y.v$ alors $t[x := u] = \lambda y.(v[x := u])$ si $x \neq y$ et y n'est pas une variable libre de u. Si y est une variable libre de u, on renomme y avant de substituer. Si $x = y$ le résultat est t.

Substitution en λ-calcul

Soient x, y, z t et a des variables, votez pour une des réponses pour chaque résultat de substitution.

- $xzty[t := ax] =$ $xzaxy$ ou $xzty$ ou ax
- $xztyt[t := ax] =$ $xzaxyt$ ou $xzaxyax$ ou $xztyt$
- $\lambda z.xz[x := yz] =$ $\lambda z.yzz$ ou $\lambda z.xz$ ou $\lambda t.yzt$
- $\lambda y.xyz[z := a] =$ $\lambda y.xya$ ou $\lambda z.xza$ ou $\lambda y.xyz$
- $\lambda y.xyz[y := a] =$ $\lambda y.xaz$ ou $\lambda y.xyz$ ou $\lambda a.xaz$

Stratégie de calcul des expressions

- Soit l'expression suivante :

$$0 * 1 + (1 - 4/3) + 3$$

Savez-vous dans quel ordre les opérateurs sont appliqués en C?

$* + -/$ | $*/ - +$ | $/ - *+$ | $+ * -/$

- Soit l'expression suivante :

$$f(e_1, e_2, ..., e_n)$$

où les e_i sont des expressions de la forme : $g_i(e'_1, ...)$.

L'ordre des applications est-il connu en C ? oui | non

S'agit-il d'un ordre total? oui | non

Le λ-calcul – la β-réduction

On appelle **rédex** un terme de la forme $(\lambda x.u)v$. On définit alors la β-réduction

$$(\lambda x.u)v \longrightarrow u[x := v]$$

- La réduction du terme $(\lambda x.u)v$ est la valeur de la fonction $\lambda x.u$ appliquée à la variable v.
- u est l'image de x par la fonction $(\lambda x.u)$,
- L'image de v est obtenue en substituant dans u, x par v.

Exemple

- $(\lambda x.xy)a$ donne $xy[x := a] = ay$
- $(\lambda x.y)a$ donne $y[x := a] = y$

Remarque Les termes sont des arbres avec des noeuds binaires (applications), des noeuds unaires (les λ-abstractions) et des feuilles (les variables). Les réductions permettent de modifier l'arbre, cependant l'arbre n'est pas forcément plus petit après l'opération. Par exemple, si l'on réduit

$$(\lambda x.xxx)(\lambda x.xxx)$$

on obtient

$$(\lambda x.xxx)(\lambda x.xxx)(\lambda x.xxx)$$

β-réductions en λ-calcul

Soient x, y, z t et a des variables, votez pour un résultat de β-réductions.

- $(\lambda x.x)\ a \longrightarrow$ $\lambda a.a$ ou a
- $(\lambda x.x)\ (\lambda y.a) \longrightarrow$ $\lambda y.a$ ou $\lambda x.x$
- $(\lambda x.x)\ ((\lambda y.a)\ z) \longrightarrow$ $(\lambda y.a)\ z$ ou a ou $\lambda x.x$
- $(\lambda x.x)\ ((\lambda y.a)\ (\lambda y.yt)) \longrightarrow$ $(\lambda x.x)\ a$ ou $(\lambda y.a)\ (\lambda y.yt)$ ou a
- $(\lambda x.(\lambda y.xay))\ z\ t \longrightarrow$ $(\lambda y.zay)\ t$ ou zat ou $(\lambda x.xaz)\ t$

Le λ-calcul – la normalisation

Un lambda-terme t est dit en forme normale si aucune β-réduction ne peut lui être appliquée, c'est-à-dire si t ne contient aucun rédex.

Remarques

- On peut simuler la normalisation des λ-termes à l'aide d'une machine de Turing, et simuler une machine de Turing par des λ-termes.
- Différentes stratégies de réduction sont définies dans le λ-calcul : stratégie applicative (par valeur, dans les langages lisp et scheme), stratégie paresseuse (par nom, dans Haskell).
- La normalisation est un calcul confluent. Soient t, u1 et u2 des lambda-termes tels que $t \longrightarrow u1$ et $t \longrightarrow u2$. Alors il existe un λ-terme v tel que $u1 \longrightarrow v$ et $u2 \longrightarrow v$. Conséquence : l'ordre d'évaluation des arguments d'une fonction n'a pas d'influence sur le résultat.

Exemple

Les symboles x, y, z, a sont des variables. Soit le terme $(\lambda x.y)((\lambda z.zz)a)$

- Stratégie applicative : $(\lambda x.y)\ \boxed{((\lambda z.zz)a)} \longrightarrow (\lambda x.y)\ \boxed{aa} \longrightarrow y$
- Stratégie paresseuse : $\boxed{(\lambda x.y)((\lambda z.zz)a)} \longrightarrow \boxed{y}$

Stratégies de réduction

- soit $f = (\lambda x.((\lambda y.y)x))$
- soit $g = (\lambda x.b)$
- soit $h = (\lambda y.y)$
- Soit le λ-terme : $(\lambda x.((\lambda y.y)x))((\lambda x.b)z)$
- C'est-à-dire $(f(gz))$

Dans les cas suivants, la stratégie est-elle Paresseuse ou Applicative ?

- Application de f à gz
- Application de g à z
- Application de h à x

Lien avec la syntaxe lisp

La syntaxe lisp est complètement basée sur le λ-calcul. Les parenthèses servent à délimiter les termes et les applications.

- Variables : x, et constantes de types numériques, symbolique, fonctionnel etc.
- Abstractions fonctionnelles : $\lambda x.y$ s'écrit (**lambda**(x) y)
- Application : uv s'écrit (u v)
  - Cas d'une abstraction fonctionnelle : ((**lambda**(x) y) a)
  - Cas d'une fonction nommée f (variable) ; fx s'écrit (f x)

Exemple

- Application d'une abstraction fonctionnelle
  - `((lambda (x) (x y)) a)` $\longrightarrow$ `(a y)`
  - `((lambda (x y) (x z y)) a b)` $\longrightarrow$ `(a z b)`
- Application d'une fonction nommée
  - Soit f la fonction `(lambda (x) (x y))`
    `(f a) = ((lambda (x) (x y)) a)` $\longrightarrow$ `(a y)`
  - Soit f la fonction `(lambda (x) (+ x 1))`, avec `+` correspondant à l'opération d'addition :
    `(f 2) = ((lambda (x) (+ x 1)) 2)` $\longrightarrow$ `(+ 2 1)` $\longrightarrow$ 3

Syntaxe et évaluation lisp

- `((lambda (z) (x z)) a)` $\longrightarrow$ `(a z)` ou `(x a)` ou incorrect
- `((lambda (z) (x z)) a x)` $\longrightarrow$ `(a z x)` ou `(x a x)` ou incorrect
- `((lambda (x) (+ 3 x)) 10)` $\longrightarrow$ `(+ 3 10)` $\longrightarrow$ 13 ou incorrect ou `(+ 10 3)` $\longrightarrow$ 13
- `((lambda (z) (lambda (x) (x z))) a)` $\longrightarrow$ `(lambda (x) (x a))` ou `(lambda (z) (a z))` ou incorrect ou `(a z)`
- `((lambda (x y) (z y)) a b)` $\longrightarrow$ `(a b)` ou `(z b)` ou incorrect

Développement incrémental

Boucle Read Eval Print : REPL

1. Read : Lecture d'une expression
2. Eval : calcul (réduction) de l'expression
3. Print : affichage du résultat (forme normale)
4. Affichage du prompt $>$ et retour à 1

- **Top-level** : niveau de la REPL, l'imbrication des expressions induit plusieurs niveaux. Par exemple pour l'expression (+ 3 4 (* 1 2) 3)
  $\longrightarrow$ évaluation de $(*\ 1\ 2)$
  $\longrightarrow$ résultat 2
  $\longrightarrow$ évaluation de (+ 3 4 3 3)
  $\longrightarrow$ résultat 13
- Notation

```
> (+ 3 4 (* 1 2) 3)
12
```

Définition et évaluation des expressions

Expressions symboliques

On appelle **expressions symboliques** (sexpr) les formes syntaxiquement correctes :

- Objet (nb, chaîne, symbole, etc.)
- Expression composée (sexpr sexpr ... sexpr) : liste de sexpr. Utilisé à la fois pour le code et les données :
  - Notation de l'application d'une fonction à ses arguments.
  - Notation des listes[a] : '(e1 e2 ... en)

[a] Pour éviter l'application et fabriquer une liste, il faut la faire précéder d'une quote

Evaluation

- Objets auto-évaluants : objet lui-même (nombres, booléens, caractères, chaînes de caractères).
- Symboles : valeur associée (identificateurs)
- Expression symbolique composée : application – évaluation de l'objet en position fonctionnelle (la première), évaluation des arguments[a], puis application de la fonction aux arguments et renvoi du résultat.

[a] dans un ordre non spécifié

Résumé des constructions syntaxiques du langage

Types

- Simples : Boolean, Number, Character, String, Symbol
- Conteneurs : Pair, List, Vector
- Fonctions : Procedure

Variables et liaisons

- Locales : **let**, **let**$*$, **letrec**, **letrec**$*$, **let**$-$**values**, **let**$*-$**values**
- Globales ou locales : **define**

Formes

Expression ou formes **define**, **let**, **lambda**, **if**, **cond**, **set**!, etc.

Expressions

constante, $(f\ a_1\ a_2\ \ldots\ a_n)$

Procédures

lambda

Macros

define$-$**syntax**

Continuations

call-with-current-continuation

Bibliothèques

library, import, export

Les types

Les booléens

- Constantes : #t et #f
- Toute valeur différente de #f est vraie
- L'objet #t est utilisé comme valeur vrai, quand aucune autre valeur ne paraît plus pertinente.
- Prédicat boolean?

Opérations booléennes

- **and** : stop au premier argument évalué à faux
- **or** : stop au premier argument évalué à vrai
- **not**
- nand, nor, xor, implies

Les opérateurs *and*, *or*, *nand*, *nor* et *xor* admettent n arguments, $n \geq 0$ et sont des formes spéciales.

Exemple

```
> (and)
#t
> (or)
#f
```

Expressions booléenes

Quels sont les résultats des expressions suivantes?

- (and 1 2) : #t ou #f ou 1 ou 2
- (and #f 2) : #t ou #f ou 2

Les types

Tour des types numériques

- Number
- Complex
- Real
- Rational
- Integer

Exactitude

- Prédicat : exact?

Les nombres

Les entiers

- Taille non limitée (seulement par la mémoire)
- Prédicat : integer?

Exemple

```
> (integer? 1)
#t
> (integer? 2.3)
#f
> (integer? 4.0)
#t
> (exact? 4)
#t
```

Les nombres

Les rationnels

- 523/123
- Accesseurs : **numerator**, **denominator**
- Prédicats : **rational**?

Les réels

- 23.2e10
- Prédicat : real ?

Exemple

```
> (real? 1)
#t
> (exact? 4.0)
#f
```

Les nombres

Les complexes

- 3+2i
- Contructeurs : make−polar, make−rectangular
- Accesseurs : real−part, imag−part, magnitude, angle
- Prédicat : **complex**?

Exemple

```
> (sqrt -1)
0+1i
> (complex? -1)
#t
> (real? 1+2i)
#f
```

Prédicats numériques

- Nombres : zero?, positive ?, negative?
- Entiers : even?, odd?
- Comparaisons : `=` `<` `<=` `>` `>=` sur les réels.
- Égalités et inégalités sur les complexes.
- Nombre d'opérandes supérieur à 2.

Exemple

```
> (= 1 2 3)
#f
> (= 1 1 1)
#t
```

Opérations numériques

Arithmétiques	$+$, $-$, $*$, $/$	n arguments réels, $n \geq 0$
Unaires	add1, sub1	incrémentation, décrémentation
	max et **min**	n arguments réels, $n > 0$
Exponentiation	**sqr**, **sqrt**, **log**	
	exp	exponentielle naturelle
	expt	base arbitraire, exposant
Modulaires	modulo	
	quotient, remainder	
	quotient/remainder	renvoie 2 valeurs
	gcd, **lcm**, **abs**	
Arrondis	**floor**, **ceiling**,	
	truncate, **round**	
Trigonométrie	**sin**, **cos**, **tan**,	
	asin, **acos**, **atan**	

Expressions booléenes

Quels sont les résultats des expressions suivantes?

- (and 1 (/ 1 0)) : 1 ou Division by zero
- (or 1 (/ 1 0)) : 1 ou Division by zero

Expressions booléenes

Quels sont les résultats des expressions suivantes?

- (and 1 (/ 1 0)) : Division by zero
- (or 1 (/ 1 0)) : 1 ou Division by zero

Expressions booléenes

Quels sont les résultats des expressions suivantes?

- (and 1 (/ 1 0)) : Division by zero
- (or 1 (/ 1 0)) : 1

Les caractères et les chaînes de caractères

Constantes

- Caractère : #\a
- Chaîne : "de␣caracteres"

Prédicats

- Type :**char**?, **string**?
- Comparaisons : **char**=?, **char**<?, **char**>?,
  string=?, **string**<?, **string**>?.

Fonctions

- Constructeurs : make−string, **string**
- Accesseurs : string−ref
- Longueur : string−length
- Conversion : number−>**string**, string−>number

Les symboles

Constantes

Ce sont à la fois des noms d'identificateurs de variables et de fonctions, et des données symboliques.

- Suite de caractères alphabétiques et numériques
- Plus les caractères suivants :

  ! $ % & * + - . / : < = > %? ~
- Échappements pour les délimiteurs[a] : |*symbol*|

[a] ( ) [ ] " , ' ` ;

Prédicats

- Type : symbol?
- Égalité : **eq**?

Conversions

- symbol->**string**, string->symbol

Les expressions conditionnelles

if

La forme **if**

(**if** ⟨*condition*⟩ ⟨*alors*⟩ ⟨*sinon*⟩)

- ⟨*condition*⟩, ⟨*alors*⟩ et ⟨*sinon*⟩ sont des expressions
- Si ⟨*condition*⟩ vaut vrai, le résultat est la valeur de l'expression ⟨*alors*⟩
- Sinon, le résultat est la valeur de l'expression ⟨*sinon*⟩

Exemple

```
> (if 1 2 3)
2
> (if (= 1 2) 3 4)
4
> (if (= 1 2) 3)
; error -> if: missing an "else" expression
```

Utilisation de l'expression if

Quels sont les résultats des expressions suivantes?

1. (if (> 1 2) 3 4) : 3 ou 4
2. (+ (if (> 1 2) 3 4) 5) : 8 ou 9 ou Error

Les expressions conditionnelles

when - unless

La forme **when**

(**when** ⟨*condition*⟩ ⟨e_1⟩ . . . ⟨e_n⟩)

Cette forme évalue les expressions ⟨e_i⟩ et renvoie le résultat de la dernière quand l'expression ⟨*condition*⟩ vaut vrai.

La forme **unless**

(**unless** ⟨*condition*⟩ ⟨e_1⟩ . . . ⟨e_n⟩)

Même chose mais quand l'expression ⟨*condition*⟩ vaut faux.

Les expressions conditionnelles

cond

La forme **cond**

(**cond** $\langle c_1 \rangle$ ... $\langle c_n \rangle$)

- Les $\langle c_i \rangle$ sont des clauses : [$\langle condition \rangle$ $\langle e_1 \rangle$ ... $\langle e_n \rangle$]
- $\langle condition \rangle$, $\langle e_1 \rangle$, ... $\langle e_n \rangle$ sont des expressions
- Évaluation des conditions des clauses dans l'ordre de $\langle c_1 \rangle$ à $\langle c_n \rangle$
- Soit c_i=[c e_1 ... e_n] la première clause dont la condition c vaut vrai, les e_i sont évaluées dans l'ordre et le résultat est celui de e_n.

Exemple

```
(cond [(number? x) "X est un nombre"]
      [(symbol? x) "X est un symbole"]
      [else (...)])
```

Les crochets définissant les clauses peuvent être remplacés par des parenthèses, conformément à la norme R6RS du langage Scheme

Utilisation de l'expression cond

Quels sont les résultats des expressions suivantes?

- (cond [(= 1 2) 1] [(< 2 3) 2] [else 3]) : 1 ou 2 ou 3
- (cond [(= 1 2) 1] [#f 2] [else 3]) : 1 ou 2 ou 3
- (cond [(= 1 2) 1] [0 2] [else 3]) : 1 ou 2 ou 3

Symboles et liaisons

Définitions

- Un **symbole** est un identificateur, c'est-à-dire un nom symbolique.
- Une **liaison** est une entité, c'est-à-dire un objet nommé résidant dans la mémoire, donc l'association d'un symbole avec un emplacement mémoire contenant une valeur.

Exemple

```
int
g(int i)
{
  return i;
}
```

```
int
f(int n)
{
  int i=1;
  return i+n;
}
```

Dans un programme un même symbole peut apparaître dans plusieurs liaisons. De même, en C, un identificateur peut aussi servir à nommer plusieurs entités. Plusieurs stratégies de recherche ont été implémentées dans les langages de programmation.

Environnements global et locaux

L'environnement est formé de **liaisons** *symbole* $\longrightarrow$ *valeur*. Les symboles ne sont pas typés (non déclarés), mais leurs valeurs le sont. Il s'agit d'un **typage dynamique**.

Environnement global : la forme **define** au top-level

- Variables : (**define** $\langle v \rangle$ $\langle e \rangle$)
- Fonctions : (**define** ($\langle f \rangle$ $\langle p_1 \rangle$ $\langle p_2 \rangle$ ... $\langle p_n \rangle$) $\langle e_1 \rangle$ $\langle e_2 \rangle$ ... $\langle e_n \rangle$)
- Résultat non spécifié par la norme

Une définition établit une liaison entre une variable et un objet résultant de l'évaluation de l'expression, cet objet pouvant être une fonction.

Exemple

```
> (define a 0)
> (define (f x) x)
> (define g (lambda(x) (* 2 x)))
```

```
> a
0
> (f 1)
1
```

```
> (g 1)
```

1 2 error

Définitions et évaluation

- Quelle est la forme correcte pour définir une fonction : `(define g(x) x)` ou `(define (f x) x)`
- Quelle est la forme correcte pour appliquer une fonction : `f(1)` ou `(f 1)`
- Le symbole *e* dans cette définition est-il relié à une fonction ou bien à un numérique ? `(define e (* 2 4))`
- Le symbole *e* dans cette définition est-il relié à une fonction ou bien à un numérique ? `(define (e) (* 2 4))`
- Soient les définitions suivantes : `(define (f) 1)` `(define (g x) (* 2 x))` Parmi les expressions suivantes lesquelles sont valides :
  - `(f 1)` `(f)` `(f 1 2)`
  - `(g 1)` `(g)` `(g #t)`