PROGRAMMATION APPLICATIVE

HLIN403

Vincent Boudet

7 janvier 2020

D'après Annie Chateau
D'après Christophe Dony

λ

Introduction

OUVRAGES LIÉS

- “Passeport pour l’algorithmique et Scheme” de R.Cori et P.Casteran.
- “Structure and Interpretation of Computer Programs” de Abelson, Fano, Sussman.
- “Programmation récursive (en Scheme)” de Christian Queinnec

PROGRAMMATION APPLICATIVE

- Programmation dans laquelle un programme est un ensemble de **définitions** de fonctions

- L'exécution d'un programme est une succession d'**applications** de fonctions à des arguments au sens algébrique du terme (calcul utilisant des opérations, des lettres et des nombres).

- Programmation où la mémoire utilisée par les calculs est automatiquement gérée par l'**interpréteur** du langage, donc sans affectation et sans effet de bord.

- **Fonction récursive :** C'est une fonction capable de s'appeler elle-même.
- Les fonctions récursives sont idéales pour implanter un algorithme pour tout problème auquel peut s'appliquer une résolution par récurrence, ou pour traiter des stuctures de données récursives (listes, arbres).
- Omniprésentes en info et dans le web

LISP, SCHEME

- Langages dédiés à la programmation applicative
- Langage de programmation universel, historiquement majeur, permettant notamment la programmation applicative.
- Syntaxe simple
- Fort pouvoir d'expression

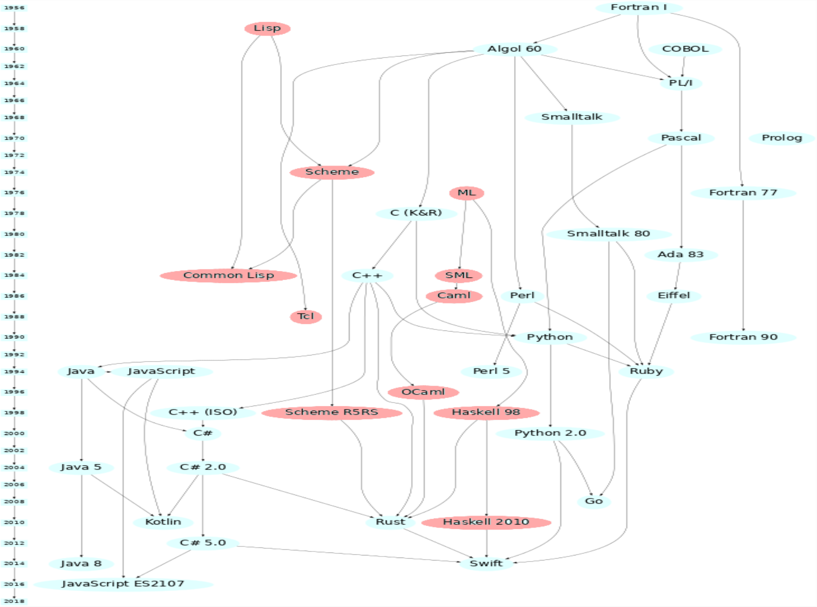
LISP, SCHEME

- **Lisp**, 1960, John Mc Carthy : pour du calcul numérique classique et du calcul symbolique (calcul dont les données ne sont pas des nombres mais des chaînes de caractères, des mots, des collections de choses).

- **Scheme**, 1970, Gerald J. Sussman et Guy L. Steele : Héritier de LISP à liaison strictement lexicale : enseignement de la programmation. “Structure and Interpretation of Computer Programs - Abselson, Fano, Sussman - MIT Press - 1984”.

- Excellent préalable à l’étude des langages à objets, plus complexes.

LISP, SCHEME ET LES AUTRES

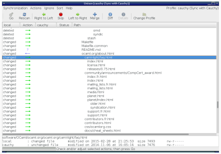
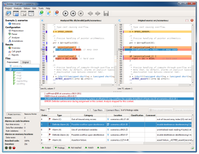


En Lisp/Scheme :

MIT APP INVENTOR

Naughty Dog, MIT App Inventor

En CAML :

Unison, Astrée, FB messenger

- Toute phrase syntaxiquement correcte du langage (y-compris une instruction) est une expression algébrique ayant une valeur. Ceci n'est pas vrai dans les langages dits ”impératifs”.
- La mémoire est gérée automatiquement (allocation et récupération) si on le souhaite (programmation sans **Effets de bords**).

Transparence référentielle : le résultat du programme ne change pas si on remplace une expression par une expression de valeur égale.

N'existe pas en C par exemple :

```c
int n = 2;
int inc(int k) {/* incrémentation par effet de bord */
  n = n + k;
  return n;
}
f(inc(1) + inc(1));
```

On ne peut remplacer **`inc(i) + inc(i)`** par **`2 * inc(i)`**

- Un très grand nombre d'abstractions simplifient la vie du programmeur.
  Exemple : les grands nombres, les itérateurs, etc.

```
> (fact 30)
265252859812191058636308480000000
> (map fact '(2 3 4 30))
(2 6 24 265252859812191058636308480000000)
> (+ 3/4 2/16)
7/8
> (* 1+i 4-i)
5+3i
```

Informatique : De « INFORmation AutoMATIQUE ». Mot inventé en 1962 par P. Dreyfus.

L'informatique est la science du traitement automatisé de l'information. Le calcul au sens arithmétique du terme est un cas particulier de traitement de l'information.

« science du traitement rationnel, notamment par machines automatiques, de l'information considérée comme le support des connaissances humaines et des communication dans les domaines techniques, économiques et sociaux » (académie française).

Traitement automatisé de l'information implique :

- codage de l'information pour sa représentation dans une machine.
- traitement de l'information selon une suite de calculs ou de transformations définis par un texte appelé programme

Données : entité manipulées dans les programmes.

programme : texte décrivant sous une forme interprétable par un ordinateur, une suite d'actions à réaliser sur des données.

Ordinateur : machine où un programme (un texte) est transformé en exécuté (transformé en actions).

Algorithme [1] : Séquence d'opérations visant à la résolution d'un problème en un temps fini (mentionner la condition d'arrêt). Fondé sur la thèse de Church.

exemple : algorithme de l'école primaire pour multiplier deux nombres.

Processus, ou processus de calcul, ou Calcul : Exécution d'un programme par une machine.

Texte d'un programme : texte, écrit dans un langage de programmation, destiné à faire exécuter des actions à un ordinateur.

1. Terme venu du XIIIème siècle, de la traduction en latin d'un mémoire de Mohammed Ibn Musa Abu Djefar Al-Khwarizmi commençant par : ≪ Algoritmi dixit... ≫.

Syntaxe : ensemble de règles de combinaison correcte des mots d'un langage pour former des phrases. (Syntaxe du français ou syntaxe de Lisp).

Grammaire : une représentation par intention, de la syntaxe d'un langage.

Analyse syntaxique, peut utiliser une grammaire, et dit si une phrase d'un langage est ou non syntaxiquement bien formée.

Semantique : ensemble de règle définissant le sens d'une phrase.

Il possible d'écrire un texte syntaxiquement correct qui n'a pas de sens selon une sémantique donnée. Exemple en français : “Quelle la différence entre un pigeon ?”

Interpréteur : programme transformant un texte de programme syntaxiquement correct en actions. L'interpréteur fait faire à l'ordinateur ce que dit le texte.

Compilateur : Traducteur de texte d'un langage L1 vers un langage L2. Par exemple, compilateur de Java en instructions de la JVM, compilateur de C en assembleur.

Modèle de calcul : modèle permettant de représenter, afin de comprendre, mesurer prouver, la façon dont un programme est exécuté.

Exemples de modèles de calcul : machine de Turing, lambda-calcul [2]

2. Church-Alonso, (14/06/1903 - 11/08/1995).

CONTENU DU COURS

- Syntaxe de *Scheme*
- Types prédéfinis. Base de l'interprétation des expressions.
- Fonctions, Identificateurs, Premières Structures de contrôle.
- Fonctions Récursives Simples.
- Listes - Symboles - Calcul Symbolique
- Fonctions récursives sur les listes. Récursions arborescentes
- Optimisation des fonctions récursives
- Abstraction de données - Arbres binaires
- Introduction à l'interprétation des langages

Scheme

PLAN DU COURS

Scheme

- Syntaxe de Scheme
- Types de données
- Interprétation et valeur d'une expression

Syntaxe : ensemble de règles de combinaison correcte des mots d'un langage pour former des phrases. (Syntaxe du français ou syntaxe de C ou de Scheme).

Les expressions de `scheme` sont appelées des **S-expressions** (expressions symboliques). Voici une définition simplifiée de leur syntaxe en forme BNF.

BNF est un acronyme pour "Backus Naur Form". John Backus et Peter Naur ont introduit cette notation formelle pour décrire la syntaxe d'un langage donné. Ce formalisme a été utilisé notamment pour Algol-60.

NOTATION BNF

: :=	se définit comme
|	ou
<symbol>	symbole non terminal
{ }	répétition de 0 à n fois
...	suite logique

SYNTAXE DES S-EXPRESSIONS

```
s_expression ::= <atom> | <expression_composee>

expression_composee ::== (s_expression s_expression)

atom ::= <atomic_symbol> | <constant>

constant ::= <number> | <litteral_constant>

atomic_symbol ::== <letter>{<atom_part>}

atom_part ::== empty | letter{atom_part} | digit{atom_part}

letter ::= "a" | "b" | ... | "z"

digit ::= "1" | "2" | ... | "9"

empty = ""
```

EXEMPLE D'EXPRESSIONS BIEN FORMÉES

```
123
true
"paul"
"coucou"
(+ 1 2)
(+ (- 2 3) 4)
(lambda (x y) (+ x y))
(sin 3.14)
```

SYNTAXE PRÉFIXÉE ET TOTALEMENT PARENTHÉSÉE

La notation mathématique n'est pas homogène.

`Scheme` propose pour l'application d'une fonction à des opérandes une syntaxe dite **préfixée** et **totalement parenthésée** :

```
(operateur opérande1 ... opérandeN)
```

- où *operateur* doit désigner une fonction connue du système
- et où les *opérandes* doivent désigner des valeurs dont les *types* sont compatibles avec la fonction.

SYNTAXE PRÉFIXÉE ET TOTALEMENT PARENTHÉSÉE

Cette syntaxe supprime les problèmes de priorité des opérateurs.

Un opérateur (une fonction) est dit *unaire*, *binaire* ou *n-aire* selon son nombre d'opérandes. Le nombre d'opérande définit l'*arité* de la fonction.

```
(+ 2 3 7)
(* (- 2 3) 4)
(f 2)
```

Donnée : entité manipulée dans un programme, définie par un type.

ex : Les entiers 1, 3 et 5 sont des données de type `integer`.

Type de donnée : entité définissant comment un ensemble de données sont représentées en machines et quelles sont les fonctions qui permettent de les manipuler.

SORTES DE TYPES

Par exemple, le type `Integer` fournit une représentation des nombres entiers (on peut les manipuler) et un ensemble de fonctions : `+`, `-`, `integer?`, etc.

Type prédéfini : Tout langage de programmation est fourni avec un ensemble de types dits types prédéfinis.

En scheme les principaux types prédéfinis sont : `integer, char, string, boolean, list, procedure, ...`

SORTES DE TYPES

Type scalaire : Type dont tous les éléments sont des constantes.

exemple : `integer, char, boolean`.

Type structuré : Type dont les éléments sont une aggrégation de données

exemple : `pair, array, vector`.

LES CONSTANTES LITTÉRALES DES TYPES PRÉDÉFINIS

Pour chaque type prédéfini d'un langage il existe des valeurs constantes qu'il est possible d'insérer dans le texte des programmes, on les appelle des constantes littérales.

ex : 123, `#\a`, "bonjour tout le monde", `#t`, `#f`

Interpréteur : programme transformant un texte de programme syntaxiquement correct en actions. L'interpréteur fait faire à l'ordinateur ce que dit le texte.

Si le texte de programme est une expression algébrique, l'interpréteur calcule sa valeur ou l'**évalue**. On peut alors parler au choix d'*interpréteur* ou d'*évaluateur*. [3]

La fonction d'évaluation est usuellement nommée `eval`.

Notation. On écrira `val(e) = v` pour signifier que la valeur d'une expression `e`, calculée par l'évaluateur, est `v`.

3. Dans le cas d'un langage non applicatif, comme Java par exemple, on parle d'interpréteur des instructions.

EVALUATION DES CONSTANTES LITTÉRALES

Soit `c` une constante littérale, `val(c) = c`.

Par exemple, `33` est un texte de programme représentant une expression valide (une constante littérale) du langage scheme, dont la valeur de type entier est *33*.

APPARTÉ : LE *TOPLEVEL*

Un toplevel d'interprétation (ou d'évaluation) est un outil associé à un langage applicatif qui permet d'entrer une expression qui est lue puis analysée syntaxiquement (lecteur), puis évaluée (évaluateur) et dont la valeur est finalement affichée (afficheur ou *printer*).

Le processus peut être décrit algorithmiquement ainsi :

```
tantque vrai faire
  print ( eval ( read ()))
fin tantque
```

et être écrit en scheme :

```
(while #t (print (eval (read))))
```

APPARTÉ : LE *TOPLEVEL*

Exemple d'utilisation du *toplevel* pour les constantes littérales :

```
> #\a
= a
> "bonjour"
= "bonjour"
> 33
= 33
```

EVALUATION D'UN APPEL DE FONCTION

Un appel de fonction se présente syntaxiquement sous la forme d'une expression composée : `(fonction argument1 argumentN)` dont la première sous-expression doit avoir pour valeur une fonction connue du système.

Exemples :

```
(sqrt 9)
```

ou

```
(+ 2 3)
```

ou

```
(modulo 12 8)
```

Valeur de l'expression `sqrt` : `<primitive:sqrt>`

ou `<primitive:sqrt>` est ce qui est affiché par la fontion *print* pour représenter la fonction prédéfinie calculant la racine carrée d'un nombre.

Ce que confirme l'expérimentation suivante avec le *toplevel* du langage :

```
> sqrt
<primitive:sqrt>
```

EVALUATION D'UN APPEL DE FONCTION

```
val ((f a1 ... aN)) = apply (val (f), val(a1), ..., val(aN))
```

où `apply` applique la fonction `val (f)` aux arguments `val(a1) ... val(aN)`

c'est-à-dire :

évalue les expressions constituant le corps de la fonction dans l'environnement constitué des liaisons des paramètres de la fonction à leurs valeurs

EVALUATION D'UN APPEL DE FONCTION

L'ordre d'évaluation des arguments n'est généralement pas défini dans la spécification du langage. Cet ordre n'a aucune importance tant qu'il n'y a pas d'**effets de bord**.

Exemples :

```
> (* 3 4)
= 12
> *
#<primitive:*>
> (* (+ 1 2) (* 2 3))
= 18
> (* (+ (* 1 2) (* 2 3)) (+ 3 3))
= 48
```

EVALUATION D'UN APPEL DE FONCTION

Evaluation en pas à pas montrant la suite des évaluations effectuées.

```
--> (* (+ 1 2) (* 2 3))

--> *

<-- #<primitive:*>

--> (+ 1 2)

--> +

<-- #<primitive:+>

--> 1

<-- 1

--> 2

<-- 2

<-- 3

--> (* 2 3)

--> *

<-- #<primitive:*>

--> 2

<-- 2

--> 3

<-- 3

<-- 6
```

ERREURS DURANT L'ÉVALUATION

```
> foo
reference to undefined identifier: foo
> (f 2 3)
reference to undefined identifier: f
> (modulo 3)
modulo: expects 2 arguments, given 1: 3
> (modulo 12 #\a)
modulo: expects type <integer> as 2nd argument, given: #\a;
> (log 0)
log: undefined for 0
```

La quatrième erreur illustre ce qu'est un langage dynamiquement typé.

Toutes les entités manipulées ont un type mais ces derniers sont testés durant l'exécution du programme (l'interprétation des expressions) et pas durant l'analyse syntaxique ou la génération de code (compilation).

Fonctions

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

Langage de programmation théorique[4] 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.

4. 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 2 (c’est-à-dire la fonction $f : x \mapsto x + 2$) 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 fonctions 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.

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 : 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.

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
```

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
```

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.

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)`

```
(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` $\mathit{val}(\mathit{test}) = \mathit{vrai}$ `alors` $\mathit{val}(\mathit{av})$ `sinon` $\mathit{val}(\mathit{af})$

NOMBRES

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

comparaisons < > `<=` `...`

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

constantes littérales : `#\a #\b`

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

tests : `char? char-numeric?`

transformation : `char-upcase`.

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)
```

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

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

Fonctions récursives

Une définition inductive d'une partie X d'un ensemble consiste à fournir :

- la donnée explicite de certains éléments de X (base) ;
- le moyen de construire de nouveaux éléments de X à partir d'éléments déjà construits.

Exemple : ensemble des valeurs de la fonction "factorielle" sur les entiers.

Les valeurs de cet ensemble peuvent être calculées par un ordinateur par exemple grace à la fonction *scheme* suivante :

```scheme
(define fact (lambda (n)
  (if (= n 0)
    1
    (* n (fact (- n 1))))))
```

DÉFINITION INDUCTIVE

Une version en C

```
int fact(int n)
{
  if (n == 0)
    return 1;
  else
    return n * fact(n-1);
}
```

Rappel **Itérer** : répéter *n* fois un processus en faisant changer la valeur des variables jusqu'à obtention du résultat.

Calcul itératif de factorielle d'un nombre : $n! = \prod_{i=1}^{n} i$

Un calcul itératif se programme par une boucle (*for* ou *tant-que* ou *repeat-until*).

Exemple de fonction itérative pour le calcul de factorielle en C

Rappel : Un invariant de boucle est une propriété qui reste vraie à chaque passage dans la boucle.

Utilité : contrôler, manuellement ou automatiquement, la bonne évolution du cacul.

```c
int fact(n) { // n entier
  int i = 0;
  int result = 1;
  while (i < n) {
    // result = fact(i) -- invariant de boucle
    i = i + 1;
    result = result * i;
    // result = fact(i) -- invariant de boucle
  }
  // i = n
  return(result);
}
```

Inconvénient : nécessité de gérer explicitement l'évolution des variables, l'ordre des affectations et le contrôle des invariants de boucle.

Autre version condensée en C :

```c
int factorielle_iterative(int n) {
  int res = 1;
  for (; n > 1; n--)
    res *= n;
  return res;
}
```

AUTRES EXEMPLES DE FONCTION RÉCURSIVES SIMPLES

Multiplication : $an = a + a(n-1)$

```
(define mult (lambda (a n)
  (= n 0)
      (a (- n 1))))))
```

AUTRES EXEMPLES DE FONCTION RÉCURSIVES SIMPLES

Puissance : $a^n = a.a^{n-1}$

```
(define exp (lambda (a n)
  (if (= n 0)
    1
    (* a (exp a (- n 1))))))
```

AUTRES EXEMPLES DE FONCTION RÉCURSIVES SIMPLES

Inverse d'une chaîne :

$inverse(n) = stringAppend(dernier(n), inverse(saufDernier(n)))$

```
(define inverse (lambda (s)
  (let ((l (string-length s)))
    (if (= 0 l)
        s
        (string-append (substring s (- l 1) l)
                       (inverse (substring s 0 (- l 1))))))))
```

CALCUL DES TERMES DE SUITES RÉCURRENTES

Suite : ensemble d'éléments indexés par N ou une partie de N.

Suite récurrente : suite dont le calcul de la valeur d'un terme peut être exprimé en fonction de la valeur des termes précédents

Suite arithmétique : $u_{n+1} = u_n + r$ ou $u_n = u_0 + nr$ suite de premier terme u_0 et de raison r.

Suite géométrique de raison q : $u_{n+1} = q.u_n$

Suite Arithmético-géométrique (ou récurrente linéaire d'ordre 1)
$u_{n+1} = \alpha u_n + \beta$

Suite récurrente linéaire d'ordre p

Toute suite à valeurs dans un corps K (généralement $\mathbb{C}$ ou $\mathbb{R}$) définie pour tout $n \geq n_0$ par la relation de récurrence suivante :

$a_0, a_1, \ldots a_p$ étant p scalaires fixés de K (a_0 non nul), pour tout $n \geq n_0$, on a

$$u_{n+p} = a_0 u_n + a_1 u_{n+1} + \cdots + a_{p-1} u_{n+p-1}$$

CALCUL DES TERMES DE SUITES RÉCURRENTES

Toute valeur d'une suite récurrente de la forme :

$u_0 = \textit{initial}$ et pour $n > 1, u_n = \Phi(u_{n-1}, n)$

peut être calculée par une fonction (de n'importe quel langage de programmation autorisant la définition de fonctions récursives) similaire à la fonction *Scheme* suivante :

```scheme
(define u (lambda (n)
  (if (= n 0)
    initial
    (PHI (u (- n 1)) n))))
```

FACTORIELLE

Par exemple calcul de factorielle de 5 :

```
(define initial 1)

(define PHI *)

(u 5) --> 120
```

Tout terme d'une suite arithmétique de raison r de la forme :

$u_0 = \textit{initial}$ et pour $n > 1, u_n = u_{n-1} + r$ peut être calculée par la fonction

```
(define ua (lambda (n r)
  (if (= n 0)
    initial
    (+ (ua (- n 1) r) r))))
```

Exemple : Multiplication de n par a,

```
(define initial 0)
(ua 3 4)
```

A noter que le code suivant ne fonctionne pas (voir cours No 3, liaison lexicale) :

```
(let ((initial 0)) (ua 3 4))
```

Pour éviter de passer par une variable globale et de rajouter un paramètre inutile, on peut utiliser la structure de contrôle `letrec`

(letrec <bindings> <body>)

Syntax: <Bindings> should have the form ((<variable1> <init1>) ...),

and <body> should be a sequence of one or more expressions. It is an error for a <variable> to appear more than once in the list of variables being bound.

Semantics: The <variable>s are bound to fresh locations holding undefined values, the <init>s are evaluated in the resulting environment (in some unspecified order), each <variable> is assigned to the result of the corresponding <init>, the <body> is evaluated in the resulting environment, and the value(s) of the last expression in <body> is(are) returned. Each binding of a <variable> has the entire letrec expression as its region, making it possible to define mutually recursive procedures.

```
(define ua (lambda (n r initial)
  (letrec ((f (lambda (n)
                (if (= n 0)
                  initial
                  (+ r (f (- n 1)))))))
          (f n))))

(ua 3 4 0)
```

= 12

Tout terme d'une suite géométrique de raison q de la forme :

$u_0 = \textit{initial}$ et pour $n > 1, u_n = q.u_{n-1}$ peut être calculée par la fonction `ug` suivante :

```
(define ug (lambda (q n initial)
  (letrec ((f (lambda (n)
              (if (= n 0)
                  initial
                  (* q (f (- n 1)))))))
          (f n)))
```

Exemple : 4 puissance 3,

```
(ug 4 3 1)
= 64
```

Exemple historique : La flêche de Zénon (ou histoire d'Achille et la tortue) n'arrive jamais à sa cible située à une distance D car la distance effectivement parcourue d est d'abord la moitié de la distance ($(1/2)D$), puis la moitié de ce qui reste $(1/4)D$ puis encore la moitié de ce qui reste, etc. Elle a parcouru à l'étape 1, $(1/2^1).D$, à l'étape 2, $(1/2^1 + 1/2^2)D$, puis à l'étape n, $(\sum_{i=1}^{n} 1/2^i)D$.

La distance parcourue par la tortue à l'étape $\mathtt{n}$ est toujours inférieure à D, quelque soit $\mathtt{n}$.

```scheme
(define sz
  (lambda (n)
    (if (= n 1)
        (/ 1 (expt 2 n))
        (+ (sz (- n 1)) (/ 1 (expt 2 n))))))
```

Plus élégant, utiliser une fonction anonyme interne pour calculer `(/1 (expt 2 n))`

```
(define sz
   (let ((f (lambda (n) (/ 1 (expt 2 n)))))
       (lambda (n)
         (if (= n 1)
             (f 1)
             (+ (f n) (sz (- n 1)))))))
```

CALCUL DE LA SOMME DES TERMES D'UNE SUITE

Généralisation au calcul de la somme des termes de toute suite u.

```
(define sommeSuite (lambda (n)
  (if (= n 0)
      (u 0)
      (+ (u n) (sommeSuite (- n 1))))))
```

A essayer avec : `(define u fact)`

Optionnel : même fonctionnalité en n'écrivant qu'une seule fonction récursive, à condition de passer la fonction du calcul d'un terme en argument. La fonction somme devient une fonctionnelle ou fonction d'ordre supérieur.

```
(define sommeSuite (lambda (n u)
    (if (= n 1)
        (u 1)
        (+ (sommeSuite (- n 1) u) (u n)))))
```

On peut par exemple écrire :

```
(sommeSuite 10 (lambda (n) (/ 1 (exp 2 n))))
```

Appel récursif : un appel récursif est un appel réalisé alors que l’interprétation d’un appel précédent de la même fonction n’est pas achevé.

L’interprétation d’une fonction récursive passe par une **phase d’expansion** dans laquelle les appels récursifs sont “empilés” jusqu’à arriver à un appel de la fonction pour lequel une **condition d’arrêt** sera vérifiée, puis par une **phase de contraction** dans laquelle les résultats des appels précédemments empilés sont utilisés.

INTERPRÉTATION D'UN APPEL À UNE FONCTION RÉCURSIVE

Première comparaison de l'interprétation des versions itératives et récursives de factorielle.

La version itérative nécessite de la part du programmeur une gestion explicite de la mémoire alors que dans la version récursive, la mémoire est gérée par l'interpréteur (usuellement via une pile).

(il faut mémoriser le fait qu'après avoir calculé `fact(4)` il faut multiplier le résultat par 5 pour obtenir `fact(5)`)

DÉCOUVERTE D’UNE SOLUTION RÉCURSIVE À DES PROBLÈMES

Disposer d’une solution récursive à un problème permet d’écrire simplement un programme résolvant (calculant quelque chose de relatif à) ce problème.

La découverte de telles solutions est parfois complexe mais rentable en terme de simplicité d’expression des programmes.

Exemple : Algorithme récursif de calcul du pgcd de deux nombres non nuls :

SI b divise a

ALORS $pgcd(a, b) = b$

SINON $pgcd(a, b) = pgcd(b, modulo(a, b))$

Implantation en Scheme :

```
(define pgcd (lambda (a b)
    (if (= b 0)
        (error "b doit être non nul")
        (let ((m (modulo a b)))
              (if (= m 0)
                  b
                  (pgcd b m))))))
```

Appel récursif non terminal : appel récursif argument d'un calcul englobant.

Exemple : l'appel récursif dans la définition de factorielle est non terminal car sa valeur est ensuite multipliée par n.

Appel récursif terminal appel récursif dont le résultat est celui rendu par la fonction contenant cet appel.

Exemple : appel récursif à `pgcd` dans la fonction précédente.

Propriété : l'interprétation d'un appel récursif terminal peut être réalisée sans consommer de pile.

Il est possible, en terme de mémoire, d'interpréter une fonction récursive terminale comme une fonction itérative car la gestion de la mémoire se déduit trivialement des transformations sur les paramètres.

Exemple canonique ”pair-impair” sur les entiers naturels

$$pair(n) = \begin{cases} \text{true si } n = 0 \\ not(impair(n-1))\text{sinon} \end{cases}$$

$$impair(n) = \begin{cases} \text{false si } n = 0 \\ not(pair(n-1))\text{sinon} \end{cases}$$

Exercice : utiliser ”letrec”.

Voir, `http://classes.yale.edu/fractals/`.

Vidéo : “Fractales à la recherche de la dimension cachée”, Michel Schwarz et Bill Jersey, 2010.

Autre cours : “Les images fractales en Scheme, Une exploration des algorithmes récursifs” - Tom Mens - University de Mons-Hainaut (U.M.H.).

Programmation avec effets de bord (impressions à l'écran) :

```
(require (lib "graphics.ss" "graphics"))

(open-graphics)

(define mywin (open-viewport "Dessin" 800 800))
```

Fonction de dessin des triangles de *Sierpinski*

```
(define s-carre (lambda (n x y cote)
  (let ((c2 (/ cote 2)))
    (if (= 0 n)
      ((draw-solid-rectangle mywin) (make-posn x y) cote cote)
      (begin
        (s-carre (- n 1) (+ x (/ cote 4)) y c2)
        (s-carre (- n 1) x (+ y c2) c2)
        (s-carre (- n 1) (+ x c2) (+ y c2) c2))))))
```

APPLICATION AU DESSIN DE FIGURES FRACTALES

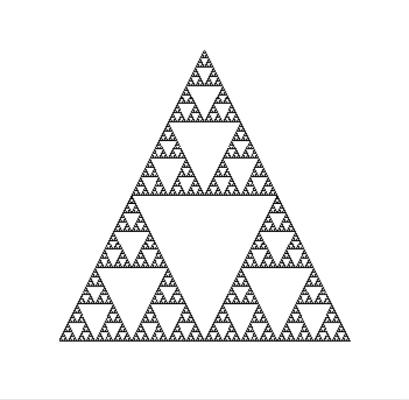


Figure 1 – `(s-carre 9 50 50 700)`

Listes et Types structurés

TYPES STRUCTURÉS

Type dont les éléments sont des agrégats de données.

- **Paire** : agrégat de deux données accessibles par leur position (première ou seconde).
- **Tableau** : agrégat d'un nombre quelconque prédéfini de données généralement homogènes (tableau d'entiers) accessibles via un index.
- **Enregistrement (record)** : agrégat de données généralement hétérogènes accessibles via un nom. Exemple, une personne est un agrégat d'un nom, d'un prénom (chaîne), d'un age (int), etc.
- **Sac** : collection quelconque non ordonnée
- **Ensemble** : collection quelconque sans ordre ni répétition.
- **Séquence - Liste** : collection ordonnée soit par l'ordre d'insertion soit par une relation entre les éléments.

LES PAIRES (DOUBLETS) EN SCHEME

Pair (paire) ou **Cons** est le type structuré de base.

Une paire, également dit “cons” (prononcer “conce”) ou doublet est un agrégat de deux données.

Exemple d’utilisation : représentation d’un nombre rationnel, agrégation d’un numérateur d’un dénominateur.

Notation : `(el1 . el2)`.

⟶ `cons`

```
(cons 1 2)
= (1 . 2)
(cons (+ 1 2) (+ 2 3))
= (3 . 5)
```

FONCTIONS D'ACCÈS

⟶ `car` : accès au premier élément.

```
(car (cons 1 2))
= 1
```

⟶ `cdr` : accès au second élément

```
(cdr (cons 1 2))
= 2
```

Une liste est une collection de données ordonnée par l'ordre de construction (ordre d'insertion des éléments).

Une liste est une **structure de donnée récursive** : une liste est soit la liste vide soit un doublet dont le second élément est une liste.

FONCTIONS DE MANIPULATION SUR LES LISTES

Les fonctions de construction et de manipulation de base sont les mêmes que celles des doublets.

```
(cons 1 (cons 2 (cons 3 ())))
= (1 . (2 . (3 . ())))
```

Notation simplifiée : `(1 2 3)`.

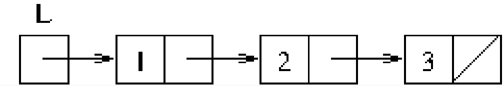


Figure 2 – Représentation interne d'une liste

car rend le premier élément du premier doublet constituant la liste donc le premier élément de la liste.

cdr rend le second élément du premier doublet c'est à dire la liste privée de son premier élément.

list est une fonction n-aire réalisant n “cons” successifs.

FONCTIONS DE MANIPULATION SUR LES LISTES

```
(cons 1 ())
= (1)
(cons 1 (cons 2 (cons 3 ())))
= (1 2 3)
(define liste2 (cons 4 (cons 6 ())))
(cons 2 liste2)
= (2 4 6)
(cons liste2 liste2)
= (((4 6) 4 6)
(list (+ 2 3) 4 6 (fact 5))
= (5 4 6 120)
```

Un programme de gestion d'un magasin doit par exemple gérer, faire des "calculs", sur des personnes, des marques, des collection de produits. Le calcul symbolique utilise des listes, des chaînes de caractères et des symboles.

Symbole : chaîne de caractères alphanumériques unique en mémoire utilisable :

- stricto sensu
- comme identificaeur (de fonction ou de variable).

SYMBOLES

- un symbole évoquant une couleur : `rouge`
- une liste évoquant une jolie maison rouge : `(une jolie maison rouge)`
- une liste de plein de choses différentes : `(jean dupond 20 marie jeanne 2 jeannette jeannot (2 bis rue des feuilles))`
- une liste d'associations : `((dupont 10) (durand 16) (martin 12))`
- une autre liste d'associations `((Hugo (ecrivain francais 19ieme)) (Moliere (...)) ...)`
- une liste représentant la définition d'une fonction scheme : `(lambda (n) (if (= n 0) 1 ...)`
- un symbole dénotant une fonction `fact`

En français mettre un texte ou un mot entre guillemets permet de faire une citation. Les guillemets empêchent le texte qu'ils délimitent d'être interprété de façon standard.

```
dis moi quel est ton age.
dis moi : "quel est ton age"!.
```

En programmation, les guillemets sont utiles dès lors qu'il y a ambiguïté possible dans l'interprétation d'une expression,

GUILLEMETS, CITATIONS

En Scheme, un appel de fonction se présente sous forme d'une liste mais toutes les listes présentes dans le texte d'un programme ne sont pas des appels de fonction.

Pour signifier à l'interpréteur qu'une expression parenthésée n'est pas un appel de fonction, il faut la mettre entre guillemets.

De même en Scheme, un identificateur est un symbole mais tous les symboles présents dans le texte d'un programme scheme ne sont pas des identificateurs. Pour indiquer à l'interpréteur qu'un symbole n'est pas un identificateur, il faut le mettre entre guillemets.

La structure de controle scheme permettant de mettre une expression entre guillements est `quote`.

Soit e une expresion de la forme `(quote exp)`, on a $val(e) = exp$.

```
(+ 2 3)
= 5

(quote (+ 2 3))
= (+ 2 3)

'(+ 2 3) //racourci syntaxique
= (+ 2 3)
```

```
(list (quote un) (quote joli) (quote chien))
= (un joli chien)

(define un 1)
(define deux 2)

(list (quote un) deux)
= (un 2)
```

- tests : `null?, list?, pair?, member?`
- renversement, concaténation.

```
(reverse '(1 2 3))
= (3 2 1)

(append '(1 2) '(3 4))
= (1 2 3 4)
```

- *reverse* d'une liste de n éléments consomme n nouveaux doublets.
- *append* de *l1* de taille *n* et de *l2* de taille *m* consomme n nouveaux doublets.

EGALITÉ PHYSIQUE ET LOGIQUE

Egalité physique : `eq?`

Egalité logique : `equal?`

A tester :

```
(equal? (list 'a 'b) (list 'a 'b))

(eq? (list 'a 'b) (list 'a 'b))
```

Extension au problème de l'appartenance à une liste : `member?` versus `memq?`

Appliquer une fonction à tous les éléments d'une liste.

```
(map number? '(1 "abc" (3 4) 5))
= (#t #f #f #t)

(map fact '(0 1 2 3 4 5))
= (1 1 2 6 24 120)
```

Il n'est pas indispensable de nommer une fonction pour la passer en argument à une fonctionnelle.

```
(map (lambda (x) x) '(a b c))

(eq? '(a b c) (map (lambda (x) x) '(a b c)))
```

Filtrer les éléments d'une liste selon un prédicat

```
(filter even? '(1 2 3 4 5))
= (2 4)

(filter number? '(1 "abc" (3 4) 5))
= (1 5)
```

Il n'est pas indispensable de nommer une fonction pour la passer en argument à une fonctionnelle.

```
(filter (lambda (x) (= (modulo x 5) 0)) '(4 5 10 12))
= (5 10)

(empty? (filter (lambda (x) (= (modulo x 5) 0)) '(4 5 10 12)))
= #f
```

Combiner les éléments d'une liste un par un

```
(foldl cons '() '(1 2 3 4))
= (4 3 2 1)

(foldl * 1 '(1 2 3 4 5))
= 120
```

Il n'est pas indispensable de nommer une fonction pour la passer en argument à une fonctionnelle.

```
(foldl (lambda (e r) (+ e r)) 0 '(1 2 3 4))
= 10

(foldl
  (lambda (val res) (+ (sqr val) res))
  0
  '(3 4))
= 25
```

Liste généralisée ou liste non plate ou arbre : liste dont chaque élement peut être une liste.

Exemple : liste d'associations.

```
(define unDico '((hugo ecrivain)
                 (pasteur médecin)
                 (knuth algorithmicien)))

(assq 'pasteur unDico)
= (pasteur médecin)
```

Arbres, voir chapitres suivants.

Exemple : Recherche d'un élément

```scheme
(define member?
  (lambda (x l)
    (cond ((null? l) #f
           ((equal? x (car l)) #t)
           (else (member? x (cdr l)))))))
```

EXEMPLE

Exemple : Recherche du nième cdr.

```
(define nthcdr
  (lambda (n l)
    (cond ((null? l) ())
          ((= n 0) l)
          (else (nthcdr (- n 1) (cdr l))))))
```

Réalisation d'un dictionnaire et d'une fonction de recherche d'une définition.

```
(define dico '((hugo lavoisier einstein joliot knuth)
  (écrivain chimiste physicien mathématicien informaticien)))

(define donneDef
  (lambda (nom)
    (letrec ((chercheDef (lambda (noms definitions)
      (cond ((null? noms) 'non-défini)
             ((eq? (car noms) nom) (car definitions))
             (#t (chercheDef (cdr noms) (cdr definitions)))))))
      (chercheDef (car dico) (cadr dico)))))

(donneDef 'joliot)
= mathématicien
```

FONCTION RÉCURSIVE ENVELOPPÉE SUR LES LISTES : SCHÉMA 1

```
(define recListe
   (lambda (l)
     (if (null? l)
       (traitementValeurArrêt)
       (enveloppe (traitement (car l))
                            (recListe (cdr l))))))
```

EXEMPLE 1

```
(define longueur
  (lambda (l)
    (if (null? l)
        0
        (+ 1 (longueur (cdr l))))))
```

- traitementValeurArret : rendre 0
- traitement du car : ne rien faire
- enveloppe : (lambda (x) (+ x 1))

EXEMPLE 1

```
(longueur '(1 3 4))

--> (+ 1 (longueur '(3 4)))

  --> (+ 1 (+ 1 (longueur '(3))))

    --> (+ 1 (+ 1 (+ 1 (longueur ()))))

      --> (+ 1 (+ 1 (+ 1 0)))

...

3
```

EXEMPLE 2

```
(define append
  (lambda (l1 l2)
    (if (null? l1)
      l2
      (cons (car l1)
            (append (cdr l1) l2)))))
```

- traitement traitementValeurArret : rendre l2
- traitement (car l) : ne rien faire
- enveloppe : cons

EXEMPLE 2

```
(append '(1 2) '(3 4))

   --> (cons 1 (append '(2) '(3 4)))

     --> (cons 1 (cons 2 (append () '(3 4))))

       --> (cons 1 (cons 2 '(3 4)))
 ...
 (1 2 3 4)
```

Consommation mémoire : taille de l1 doublets.

EXEMPLE 3

```
(define add1
  (lambda (l)
    (if (null? l)
        ()
        (cons (+ (car l) 1) (add1 (cdr l))))))
```

```
(add1 '(1 2 3))
= (2 3 4)
```

- traitementValeurArret : rendre ()
- traitement (car l) : +1
- enveloppe : cons

EXEMPLE 4 : TRI PAR INSERTION

```
(define tri-insertion
  (lambda (l)
    (letrec
      ((insertion (lambda (x l2)
        (cond ((null? l2) (list x))
          ((< x (car l2)) (cons x l2))
          (#t (cons (car l2) (insertion x (cdr l2))))))))
      (if (null? l)
        ()
        (insertion (car l) (tri-insertion (cdr l)))))))
```

Consommation mémoire : chaque insertion consomme en moyenne $taille(l2)/2$ doublets ; il y a $taille(l)$ insertions. La consommation mémoire est donc en $O(l^2/2)$

Consommation pile : pour `tri-insertion`, $taille(l)$. Pour `insertion`, en moyenne $taille(l2)/2$

FONCTION RÉCURSIVE ENVELOPPÉE - SECOND SCHÉMA

```
(define recListe2
  (lambda (l)
    (if (null? l)
      (traitement ())
      (enveloppe (recListe2 (cdr l))
                 (traitement (car l))))))
```

```scheme
(define reverse
  (lambda (l)
    (if (or (null? l) (null? (cdr l)))
        l
        (append (reverse (cdr l)) (list (car l))))))
```

Consommation mémoire : taille de l doublets.

Fonction récursive arborescente : fonctions récursives contenant plusieurs appels récursifs, éventuellement enveloppés, ou dont l'enveloppe est elle-même un appel récursif

Fonction dont l'interprétation nécessite un arbre de mémorisation.

Exemple du calcul des nombres de Fibonacci.

Problème initial : “Si l'on possède initialement un couple de lapins, combien de couples obtient-on en n mois si chaque couple engendre tous les mois un nouveau couple à compter de son deuxième mois d'existence”.

Ce nombre est défini par la suite récurrente linéaire de degré 2 suivante :

$F_0 = 0$, $F_1 = 1$, $F_n = F_{n-1} + F_{n-2}$

Les nombres de Fibonacci sont célèbres en arithmétique et en géométrie pour leur relation avec le nombre d'or, solution de l'équation $x^2 = x + 1$. La suite des quotients de deux nombres de Fibonacci successifs a pour limite le nombre d'or.

Les valeurs de cette suite peuvent être calculées par la fonction scheme suivante.

```
(define fib
  (lambda (n)
    (cond ((= n 0) 0)
          ((= n 1) 1)
          (#t (+ (fib (- n 1)) (fib (- n 2)))))))
```

Complexité :

L'interprétation de cette fonction génère un arbre de calcul (récursivité arborescente). L'**arbre de calcul** de `(fib n)` possède *fib*(n) feuilles.

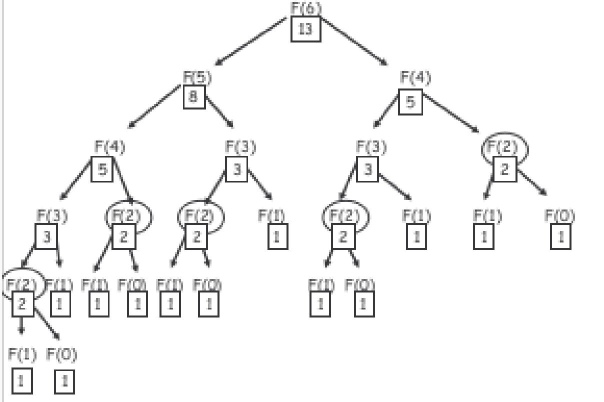


Figure 3 – Exemple, fib(6) : 13 feuilles

Complexité :

Ceci signifie que l'on calcule $fib(n)$ fois $fib(0)$ ou $fib(1)$, et que l'on effectue $fib(n) - 1$ additions.

Par exemple, pour calculer $fib(30) = 842040$, cette fonction effectue $fib(30) - 1$ soit 842039 additions.

Recherche d'un élément dans une liste généralisée (une liste contenant des listes)

```
(define genmember?
  (lambda (x lgen)
    (cond ((null? lgen) #f)
          ((equal x (car lgen)) #t)
          ((list? (car lgen))
           (or (genmember? x (car lgen))
               (genmember? x (cdr lgen))))
          (#t (genmember? x (cdr lgen))))))
```

EXERCICE

Écrire une fonction qui “aplatit” une liste généralisée.

```
(flat '((1 2) 3 (4 (5 6) 7)))
= (1 2 3 4 5 6 7)
```

Problème

L'exemple suivant, tiré de "structure and interpretation of computer programs" montre la réduction d'un problème par récursivité.

*P*roblème : soit à calculer le nombre `N` de façons qu'il y a de rendre une somme `S` en utilisant `n` types de pièces.

Il existe une solution basée sur une réduction du problème jusqu'à des conditions d'arrêt qui correspondent au cas où la somme est nulle ou au cas où il n'y plus de types de pièces à notre disposition.

RÉDUCTION DU PROBLÈME

Soient

- V l'ensemble ordonné des valeurs des différentes pièces,
- n le nombre de types de pièces (égal au cardinal de V),
- n_i le ième type de pièce
- v_i la valeur du ième type de pièce.

Par exemple, avec l'euro $V = \{1, 2, 5, 10, 20, 50, 100, 200\}$ et $n = 8$ (il y a 8 types de pièces de valeurs respectives 1, 2, 5, 10, 20, 50, 100 et 200 cts)

RÉDUCTION DU PROBLÈME

On peut réduire le problème ainsi :

$$\boxed{NFRendre(S, n) = NFRENDRE(S - v_1, n) + NFRendre(S, n - 1)}$$

VALEURS INITIALES

si $S = 0$, 1
si $S < 0$, 0 (on ne peut pas rendre une somme négative - ce cas ne peut se produire que pour des monnaies qui ne possèdent pas la pièce de 1 centime)
si $S > 0, n = 0$, 0 (aucune solution pour rendre une somme non nulle sans pièce)

EXEMPLE

```
Rendre 5 centimes avec (1 2 5) =
  Rendre 4 centimes avec (1 2 5)
  + Rendre 5 centimes avec (2 5)
soit en développant le dernier, =
  Rendre 4 centimes avec (1 2 5) ...
  + Rendre 3 centimes avec (2 5)
    + Rendre 5 centimes avec (5)
soit en développant le dernier, =
  Rendre 4 centimes avec (1 2 5) ...
  + Rendre 3 centimes avec (2 5) ...
    + Rendre 0 centimes avec (5)
      + Rendre 5 centimes avec ()
soit en appliquant les tests d'arrêt, =
  Rendre 4 centimes avec (1 2 5) ...
  + Rendre 3 centimes avec (2 5) ...
    + 1
      + 0
= 4 ((1 1 1 1 1) (1 1 1 2) (1 2 2) (5))
```

Soient :

- S la somme à rendre
- V l'ensemble ordonné des valeurs des différentes pièces
- n le nombre de types de pièces (égal au cardinal de V)
- n_i le ième type de pièce et v_i la valeur de ce ième type de pièce.

ALGORITHME

NFRendre(S, n, V) =

si $S = 0$ alors 1

sinon si $S < 0$ alors 0 (on ne peut pas rendre une somme négative)

sinon si $n = 0$ alors 0 (on ne peut pas rendre une somme sans pièce)

sinon $NFRendre(S - v_1, n, V) + NFRendre(S, n - 1, V \backslash v_1)$

IMPLANTATION

```scheme
(define NFRendre
  (lambda (somme nbSortesPieces valeursPieces)
    (letrec ((rendre (lambda (somme nbSPieces)
             (cond ((= somme 0) 1)
                   ((or (< somme 0) (= nbSPieces 0)) 0)
                   (else (+ (rendre somme (- nbSPieces 1))
               (rendre (- somme (valeursPieces nbSPieces))
                       nbSPieces)))))))
      (rendre somme nbSortesPieces))))
```

IMPLANTATION

```
(define (valeurPiecesEuro piece)
  (cond ((= piece 1) 1)
        ((= piece 2) 2)
        ((= piece 3) 5)
        ((= piece 4) 10)
        ((= piece 5) 20)
        ((= piece 6 ) 50)
        ((= piece 7 ) 100)
        ((= piece 8 ) 200)
        ))
```

IMPLANTATION

```scheme
(define (valeurPiecesDollar piece)
  (cond ((= piece 1) 1)
        ((= piece 2) 5)
        ((= piece 3) 10)
        ((= piece 4) 25)
        ((= piece 5) 50)
        ))

(define rendreEuro
  (lambda (somme)
    (NFRendre somme 8 valeurPiecesEuro)))

(define rendreDollar
  (lambda (somme)
    (NFRendre somme 5 valeurPiecesDollar)))
```

EXÉCUTION

- 4 façons de rendre 5 centimes ((1 1 1 1 1) (1 1 1 2) (1 2 2) (5)),
- 11 façons de rendre 10 centimes,
- 4112 façons de rendre 100 centimes,
- 73682 façons de rendre 200 centimes.

EXERCICE

Écrivez une variante du programme qui rend la liste des solutions.

La complexité est du même ordre que celle de la fonction fibonacci mais pour ce problème il est beaucoup plus difficile d'écrire un algorithme itératif.

Optimisation de fonctions récursives

Optimisation de fonctions récursives

Simplification de certaines fonctions récursives

Processus de calcul itératif : processus basé sur une suite de transformation de l'état de l'ordinateur spécifié par au moins une boucle et des affectations.

Affectation Instruction permettant de modifier la valeur stockée à l'emplacement mémoire associé au nom de la variable.

```
x := 12;       ;; en Algol, Simula, Pascal
x = 12;        ;; en C, Java, C++
(set! x 12)    ;; en scheme
(set! x (+ x 1))
```

UNE STRUCTURE DE CONTRÔLE POUR ÉCRIRE LES BOUCLES EN SCHEME

```
(do (initialisation des variables)
    (condition-d'arret
       expression-si-condition-d'arret-vérifiée)
  (instruction 1)
  ...
  (instruction n))
```

EQUIVALENCE ITÉRATION - RÉCURSIONS TERMINALES

Du point de vue de la quantité d'espace pile consommé par l'interprétation, les deux fonctions suivantes sont équivalentes.

```
(define ipgcd
  (lambda (a b)
     (do ((m (modulo a b)))
       ((= m 0) b)
       (set! a b)
       (set! b m)
       (set! m (modulo a b)))))
```

```
(define pgcd
  (lambda (a b)
    (let ((m (modulo a b)))
      (if (= m 0)
          b
          (pgcd b m))))))
```

Il n'est pas utile de vouloir à tout prix dérécursiver. La taille des mémoires et l'efficacité des processeurs rendent de nombreuses fonctions récursives opérantes.

Cependant certaines transformations sont simples et il n'est pas coûteux de les mettre en oeuvre. Par ailleurs les récursions à complexité exponentielles doivent être simplifiées quand c'est possible.

Pour dérécursiver, il y a deux grandes solutions : soit trouver une autre façon de poser le problème soit, dans le cas général, passer le calcul d'enveloppe (ce qu'il y a à faire une fois que l'appel récursif est achevé) en paramètre de l'appel récursif.

Une version itérative (en C)

```c
int fact(n){
  int cpt = 0;
  int acc = 1;
  while (cpt < n){
    // acc = fact(cpt) -- invariant de boucle
    cpt = cpt + 1;
    acc = acc * cpt;
    // acc = fact(cpt) -- invariant de boucle
  }
  // en sortie de boucle, cpt = n, acc = fact(n)
  return(acc)
}
```

Une version itérative (en scheme)

```
(define ifact
  (lambda (n)
    (do ((cpt 0) (acc 1))
      ;; test d'arret et valeur rendue si le test est #t
      ((= cpt n) acc)
      ;; acc = fact(cpt) -- invariant de boucle
      (set! cpt (+ cpt 1))
      (set! acc (* acc cpt))
      ;; acc = fact(cpt) -- invariant de boucle
)))
```

VERSION RÉCURSIVE TERMINALE (EN SCHEME)

Solution générale à la dérécursivation : passer le calcul d'enveloppe en argument de l'appel récursif.

Exemple de factorielle : deux paramètres supplémentaires qui vont prendre au fil des appels récursifs les valeurs successives de `cpt` et `acc` dans la version itérative.

VERSION RÉCURSIVE TERMINALE (EN SCHEME)

```scheme
(define ifact
  (lambda (n cpt acc)
    ;; acc = fact(cpt)
    (if (= cpt n)
       acc
       (ifact n (+ cpt 1) (* (+ cpt 1) acc)))))
```

Il est nécessaire de réaliser l'appel initial correctement : `(ifact 7 0 1)`

Version plus élégante, qui évite le passage du paramètre `n` à chaque appel récursif et évite le contrôle de l'appel initial.

```
(define ifact
  (lambda (n)
    (letrec ((boucle (lambda (cpt acc)
                 ;; acc = fact(cpt)
                 (if (= cpt n)
                    acc
                    (boucle (+ cpt 1) (* (+ cpt 1) acc))))))
      (boucle 0 1))))
```

VERSION RÉCURSIVE TERMINALE (EN SCHEME)

La multiplication étant associative, on peut effectuer les produits de n à 1. Ce qui donne une version encore plus simple.

```
(define ifact
  (lambda (n)
    (letrec ((boucle (lambda (cpt acc)
               ;; acc = fact(n-cpt)
               (if (= cpt 0)
                   acc
                   (boucle (- cpt 1) (* cpt acc))))))
      (boucle n 1))))
```

Trouver ce que calcule la fonction interne boucle revient à trouver l'invariant de boucle en programmation impérative. A chaque tour de boucle, on vérifie pour cette version que $acc = fact(n - cpt)$.

```
(define fib_env (lambda (n)
  (cond ((= n 0) 1)
        ((= n 1) 1)
        (#t (+ (fib_env (- n 1)) (fib_env (- n 2)))))))

(time (fib_env 10))
cpu time: 0 real time: 0 gc time: 0
89
(time (fib_env 30))
cpu time: 138 real time: 139 gc time: 0
1346269
(time (fib_env 40))
cpu time: 14482 real time: 14476 gc time: 28
165580141
(time (fib_env 100))
;; ne finit pas
```

```
(define do-fib (lambda (n)
    (do ((m n) (a 1) (b 1))
        ((or (= m 0) (= m 1)) b)
        (set! b (+ a b))
              (set! a (- b a))
              (set! m (- m 1)))))

(time (do-fib 10))
cpu time: 0 real time: 0 gc time: 0
89
(time (do-fib 30))
cpu time: 0 real time: 0 gc time: 0
1346269
(time (do-fib 40))
cpu time: 0 real time: 0 gc time: 0
165580141
(time (do-fib 100))
cpu time: 0 real time: 0 gc time: 0
573147844013817084101
```

Plus fort !

```
(time (do-fib 10000))
cpu time: 30 real time: 29 gc time: 22
544383731135652813387342609937503801353891845546959670262477158412085828
;; 2090 chiffres
```

Transformation du problème : utiliser des variables ou la pile d'exécution pour conserver en mémoire les deux dernières valeurs qui sont suffisantes pour calculer la suivante.

Observons les valeurs successives de deux suites :

$a_n = b_{n-1}$ avec $a_0 = 1$

et

$b_n = a_{n-1} + b_{n-1}$ avec $b_0 = 1$

On note que pour tout n, $a_n = fib(n)$.

On en déduit la fonction récursive suivante :

COMPARAISON DES VERSIONS (EN SCHEME) : VERSION TERMINALE

```
(define fib_term (lambda (n)
  (letrec ((aux (lambda (n a b)
                (cond ((= n 0) a)
                      (#t (aux (- n 1) b (+ a b)))))))
    (aux n 1 1))))

(time (fib_term 10))
cpu time: 0 real time: 0 gc time: 0
89
(time (fib_term 30))
cpu time: 0 real time: 0 gc time: 0
1346269
(time (fib_term 40))
cpu time: 0 real time: 0 gc time: 0
165580141
(time (fib_term 100))
cpu time: 0 real time: 0 gc time: 0
573147844013817084101
```

Au moins aussi fort !

```
(time (fib_term 10000))
cpu time: 27 real time: 27 gc time: 16
544383731135652813387342609937503801353891845546959670262477158412085
```

RÉCURSION VERS ITÉRATION, LE CAS DES LISTES

Mêmes principes que précédemment, les opérateurs changent.

```
(define longueur
  (lambda (l)
    (letrec ((boucle (lambda (l1 acc)
        ;; acc = taille(l) - taille(l1)
        (if (null l1)
            acc
            (boucle (cdr l1) (+ 1 acc))))))
      (boucle l 0))))
```

Somme des éléments d'une liste

Version explicitement itérative :

```
(define do-somme-liste
  (lambda (l)
    (do ((l1 l) (acc 0))
        ((null l1) acc)
        (set! accc (+ acc (car l)))
        (set! l1 (cdr l1)))))
```

Version récursive terminale :

```
(define isomme-liste
  (lambda (l)
    (letrec ((boucle (lambda (l acc)
                         (if (null l) acc
                     (boucle (cdr l) (+ (car l) acc))))))
      (boucle l 0))))
```

Renversement d'une liste

Version explicitement itérative.

```
(define (do-reverse l)
  (do ((current l) (result ()))
    ((null? l) result)
    (set! result (cons (car l) result))
    (set! l (cdr l))
    ;; invariant :
    ;; result == reverse (initial(l) - current(l))
    ))
```

Version récursive terminale. L'accumulateur est une liste puisque le résultat doit en être une. A surveiller le sens dans lequel la liste se construit par rapport à la version récursive non terminale.

```
(define ireverse
  (lambda (l)
    (letrec ((boucle (lambda (l acc)
              (if (null? l)
                  acc
                  (boucle (cdr l) (cons (car l) acc))))))
      (boucle l ()))))
```

Etude mathématique ou informatique du problème.

Exemple avec la fonction puissance : amélioration des formules de récurrence conjuguée à une dérécursivation.

- Version récursive standard. Cette version réalise n multiplications par x et consomme n blocs de pile.

```
(define puissance
  (lambda (x n)
    (if (= n 0)
      1
      (* x (puissance x (- n 1))))))
```

- Une version récursive terminale conforme au schéma de dérécursivation précédent nécessite toujours *n* multiplications par *x* mais ne consomme plus de pile.

```
(define puissance-v2
  (lambda (x n)
    (letrec ((boucle (lambda (cpt acc)
                       ;; puis(x,n-cpt) = acc
                       (if (= cpt 0)
                           acc
                           (boucle (- cpt 1) (* acc x))))))
      (boucle n 1))))
```

AUTRES AMÉLIORATIONS DE FONCTIONS RÉCURSIVES

- Une version récursive dite “dichotomique” nécessitant moins de multiplications. Elle est basée sur la propriété suivante de la fonction *puissance* :
  si n est pair, $\exists m = n/2$ et $x^n = x^{2m} = (x^2)^m$,
  si n est impair, $x^n = x^{2m+1} = x.x^{2m} = x.(x^2)^m$,

```scheme
(define puissance-v3
  (lambda (x n)
    (cond ((= n 0) 1)
      ((even? n) (puissance-v3 (* x x) (quotient n 2)))
      ((odd? n) (* x (puissance-v3 (* x x) (quotient n 2)))))))
```

Cette version [5] n’est pas récursive terminale quand n est impair, par contre elle n’effectue plus que de l’ordre de $log(n)$ multiplications.

5. Attention, utiliser “quotient” plutôt que “/” car quotient rend un entier compatible avec les fonctions “even” et “odd”.

- Couplage des deux améliorations. Une version itérative de la fonction puissance dichotomique suppose à nouveau le passage par un accumulateur passé en paramètre.

$f(x, 0, acc) = acc * x^0 = acc$

Si n est pair, $f(x, n, acc) = x^n.acc = (x^2)^{n/2}.acc = f(x^2, n/2, acc)$

Si n est impair, $f(x, n, acc) = x^n.acc = ((x^2)^{n/2}).(x.acc) = f(x^2, n/2, x.acc)$

On en déduit la version itérative et optimisée (dichotomie) suivante :

```
(define puissance-v4
  (lambda (x n)
    (letrec ((boucle (lambda (x n acc)
        (cond ((= n 0) acc)
          ((even? n)
              (boucle (* x x) (quotient n 2) acc))
          ((odd? n)
              (boucle (* x x) (quotient n 2) (* x acc)))))))
     (boucle x n 1))))
```

UNE SOLUTION EN $0(N)$: LA “MÉMOIZATION”

Memoization :

Du latin “memorandum”. “In computing, memoization is an optimization technique used primarily to speed up computer programs by storing the results of function calls for later reuse, rather than recomputing them at each invocation of the function” - Wikipedia

UNE SOLUTION EN $O(N)$: LA "MÉMOIZATION"

```scheme
(define memo-fib
  (lambda (n)
    (define val (lambda (n memoire)
        (cadr (assq n memoire))))
    (define memo (lambda (n memoire)
      ;; rend une liste où la valeur de fib(n) est stockée
        (let ((dejaCalcule (assq n memoire)))
          (if dejaCalcule
            memoire
            (let ((memoire (memo (- n 1) memoire)))
              (let ((fibn (+ (val (- n 1) memoire)
                             (val (- n 2) memoire))))
                ;; on ajoute à la liste memoire la dernière
                ;; valeur calculée et on la rend
                (cons (list n fibn) memoire)))))))

    (val n (memo n '((1 1) (0 1))))))
```

Abstraction

Abstraction

Abstraction de données

Programmation par flux de données (dataflow)

Donnée : entité manipulée dans un programme, définie par un type.

ex : Les entiers 1, 3 et 5 sont des données d'un programme réalisant des calculs.

Abstraction de données : représentation interne de données d'un certain type accessible via un ensemble de fonctions constituant son interface.

Interface : ensemble de fonction de manipulation de données d'un certain type.

Type Abstrait : Type pour lequel on ne peut accéder à la représentation interne des données.

Pascal : *Enregistrements - Records*

C : *Structures - Structs*

JAVA : *Classes - Class* - Réalisation de l'encapsulation “données - fonctions”.

UN EXEMPLE : LES NOMBRES RATIONNELS (D'APRÈS ABELSON FANO SUSSMAN)

Interface :

On peut distinguer dans l'interface la partie *créationnelle*, permettant de créer des données, de la partie *fonctionnelle* ou *métier* permettant de les utiliser.

La programmation par objet a généralisé la création de nouveaux types de données. L'interface créationnelle est constituée de la fonction `new` et d'un ensemble de fonctions d'initialisation (généralement nommées *constructeurs*).

UN EXEMPLE : LES NOMBRES RATIONNELS (D'APRÈS ABELSON FANO SUSSMAN)

Exemple, interface pour les nombres rationnels :

```
make-rat   ; createur
denom      ; accesseur
numer      ; accesseur
+rat       ; interface fonctionnelle
-rat
*rat
/rat
=rat
```

Implantation

```
(define +rat (lambda (x y)
    (make-rat (+ (* (numer x) (denom y))
                 (* (denom x) (numer y)))
              (* (denom x) (denom y)))))

(define *rat (lambda (x y)
    (make-rat (* (numer x) (numer y))
              (* (denom x) (denom y)))))
```

INTERFACE DE CRÉATION, REPRÉSENTATION NO 1 : DOUBLETS, SANS RÉDUCTION

```
(define make-rat (lambda (n d) (cons n d)))
(define numer (lambda (r) (car r)))
(define denom (lambda (r) (cdr r)))
```

LES DIFFÉRENTS NIVEAUX D'ABSTRACTION

```
utilisation des rationnels
------- +rat -rat make-rat numer denom

représentation des rationnels
------- doublets : cons, car, cdr

représentation des doublets
------- ???
```

INTERFACE DE CRÉATION, REPRÉSENTATION NO 2, VECTEURS ET RÉDUCTION

Vecteur : Collection dont les éléments peuvent être rangés et retrouvés via un index, qui occupe moins d'espace qu'une liste contenant le même nombre d'éléments.

Manipulation de vecteurs.

```
(make-vector taille) : création vide
(vector el1 ... eln) : définition par extension
(vector-ref v index) : accès indexé en lecture
(vector-set! v index valeur) : accès indexé en écriture
```

INTERFACE DE CRÉATION, REPRÉSENTATION NO 2, VECTEURS ET RÉDUCTION

```
(define make-rat (lambda (n d)
  (let ((pgcd (gcd n d)))
    (vector (/ n pgcd) (/ d pgcd)))))

(define numer (lambda (r) (vector-ref r 0)))

(define denom (lambda (r) (vector-ref r 1)))
```

Graphe : ensemble de sommets et d'arêtes

Graphe orienté : ensemble de sommets et d'arcs (arête “partant” d'un noeud et “arrivant” à un noeud)

Arbre : graphe connexe sans circuit tel que si on lui ajoute une arête quelconque, un circuit apparaît. Les sommets sont appelés **noeuds**.

Exemple d'utilisation : Toute expression scheme peut être représentée par un arbre.

Arbre binaire :

- soit un arbre vide

- soit un noeud ayant deux descendants (fg, fd) qui sont des arbres binaires

Arbre binaire de recherche : arbre dans lesquels une valeur v est associée à chaque noeud n et tel que si $n_1 \in fg(n)(resp.fd(n))$ alors $v(n_1) < v(n)$ (resp. $v(n_1) > v(n)$)

Arbre partiellement équilibré : la hauteur du SAG et celle du SAD diffèrent au plus de 1.

- Création :

```
(make-arbre v fg fd)
```

- Accès aux éléments d'un noeud n :

```
(val-arbre n), (fg n), (fd n)
```

- Test

```
(arbre-vide? n)
```

- Interface métier :
  - `(inserer valeur a)` : rend un nouvel arbre résultat de l'insertion de la valeur n dans l'arbre a. L'insertion est faite sans équilibrage de l'arbre.
  - `(recherche v a)` : recherche dichotomique d'un élément dans un arbre
  - `(afficher a)` : affichage des éléments contenus dans les noeuds de l'arbre, par défaut affichage en profondeur d'abord.

Consommation mémoire, feuille et noeud : 3 doublets.

Simple à gérer.

```
(define make-arbre (lambda (v fg fd)
  (list v fg fd)))

(define arbre-vide? (lambda (a) (null? a)))

(define val-arbre (lambda (a) (car a))) ;;

(define fg (lambda (a) (cadr a)))

(define fd (lambda (a) (caddr a)))
```

INSERTION SANS DUPLICATION

```
(define insere (lambda (n a)
  (if (arbre-vide? a)
      (make-arbre n () ())
      (let ((valeur (val-arbre a)))
        (cond ((< n valeur) (make-arbre valeur
                               (insere n (fg a)) (fd a)))
              ((> n valeur) (make-arbre valeur
                               (fg a) (insere n (fd a))))
              ((= n valeur) a))))))
```

```
(define a (make-arbre 6 () ()))
(insere 2 (insere 5 (insere 8 (insere 9 (insere 4 a)))))
```

Fonction booléenne de recherche dans un arbre binaire. La recherche est dichotomique.

```
(define recherche (lambda (v a)
  (and (not (arbre-vide? a))
      (let ((valeur (val-arbre a)))
        (cond ((< v valeur) (recherche v (fg a)))
              ((> v valeur) (recherche v (fd a)))
              ((= v valeur) #t))))))
```

SECONDE REPRÉSENTATION INTERNE

Coût mémoire, noeud (3 doublets), feuilles (1 doublet).

Nécessite un test supplémentaire à chaque fois que l'on accède à un fils.

```scheme
(define make-arbre (lambda (v fg fd)
  (if (and (null? fg) (null? fd))
      (list v)
      (list v fg fd))))

(define fg (lambda (n) (if (null? (cdr n)) () (cadr n))))

(define fd (lambda (n) (if (null? (cdr n)) () (caddr n))))
```

Toutes les autres fonctions de manipulation des arbres binaires (interface fonctionnelle) sont inchangées.

Le type dictionnaire est un type récurrent en programmation, on le trouve dans la plupart des langages de programmation (java.util.Dictionary).

En scheme, il a été historiquement proposé sous le nom de listes d'association. Une liste d'associations est représentée comme une liste de doublets ou comme une liste de liste à deux éléments.

```
>(define l1 '((a 1) (b 2)))
>(assq 'a l1)
(a 1)
>(assq 'c l1)
#f
>(assoc 'a l1)
(a 1)
>(define l2 '( ((a) (une liste de un élément))
               ((a b) (deux éléments)) ))
>(assq '(a b) l2)
#f
>(assoc '(a b) l2)
((a b) (deux éléments))
```

INTERFACE : CONSTRUCTION ET MANIPULATIONS

Scheme n'a pas prévu d'interface de construction, on construit directement les listes.

Ceci implique que l'on ne peut pas changer la représentation interne des dictionnaires.

Imaginons une interface de construction

```
(define prem-alist car)
(define reste-alist cdr)
(define null-alist? null?)
(define make-alist list)
(define add (lambda (clé valeur l)
               (cons (list clé valeur) l)))
```

```
(define my-assoc (lambda (clé al)
 (letrec ((loop (lambda (alis)
    (cond ((null-alist? alis) #f)
          ((equal? (car (prem-couple alis)) clé) )
                   (prem-couple alis))
          (else (loop (reste-alist alis)))))))
    (loop al))))
```

Donnée : entité manipulée dans un programme, définie par un type.

Aujourd'hui, on parle de To de données comme on parlait de Mo il y a quelques années.

Stockages de plusieurs Po accessibles

Besoin de gestion efficace et rapide (traitements, données biologiques, physiques, ou encore données internet pour analyses commerciales et autres).

$\Rightarrow$ Besoin d'un paradigme qui combine l'utilisation des processeurs multicœurs et les clusters de nombreux nœuds avec le traitement et l'analyse des données.

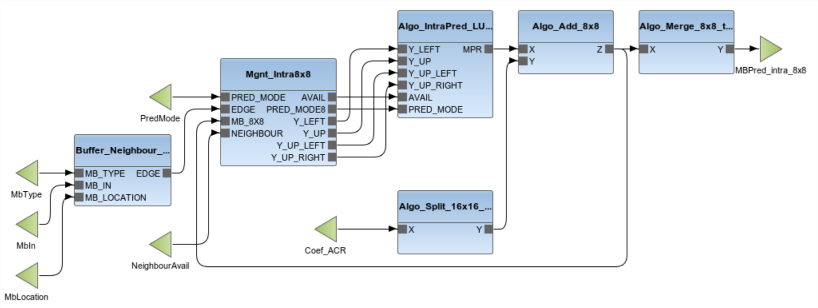
Le paradigme « flux de données » est constitué de noeuds dans un graphe orienté, reliés par des files d'attente.

Le graphe représente une application ou un programme et les noeuds représentent des fonctions qui doivent être appliquées aux données.

Les files d'attente transportent les données entre les fonctions connectées.

Dans la plupart des cas, chaque opérateur dans le graphe reçoit des données à partir de sa file d'attente d'entrée, applique sa fonction aux données, et délivre en sortie les résultats à sa file d'attente de sortie.

DATAFLOW



Algo_IntraPred_LU...
Y_LEFT
MPR
Y_UP
Y_UP_LEFT
Y_UP_RIGHT
AVAIL
PRED_MODE
Algo_Add_8x8
X
Z
Y
Algo_Merge_8x8_t...
X
Y
MBPred_intra_8x8
Mgnt_Intra8x8
PRED_MODE
AVAIL
EDGE
PRED_MODE8
MB_8X8
Y_LEFT
NEIGHBOUR
Y_UP
Y_UP_LEFT
Y_UP_RIGHT
PredMode
Buffer_Neighbour_...
MB_TYPE
EDGE
MB_IN
MB_LOCATION
MbType
MbIn
MbLocation
NeighbourAvail
Algo_Split_16x16_...
X
Y
Coef_ACR

PROGRAMMATION PAR FLUX DE DONNÉES, PRINCIPE DE BASE

La composition d'un graphe de flux de données

= processus de création d'instances d'opérateurs et de leurs liaisons entre elles par « couture » de la sortie d'un opérateur à l'entrée d'un autre.

Souplesse dans la façon dont les utilisateurs relient les opérateurs pour créer des applications.

En pensant à chaque opérateur en termes de

- consommer des données et
- produire une sortie,

le paradigme dataflow fournit le mécanisme d'ordre supérieur (composition) pour l'organisation de ces fonctions.

EXEMPLE : LE PIPE SHELL UNIX

En UNIX les commandes shell peuvent être raccordées l'une à l'autre pour produire la sortie désirée.

Chaque commande shell est indépendante mais toutes vont consommer l'entrée standard et envoient des données sur la sortie standard.

Opérateur |

```
cat "monfichier.txt" | grep "toto" | wc -l
```

FILES D’ATTENTES

Les opérateurs Dataflow utilisent les files d’attente.

Ils peuvent avoir plusieurs files d’attente d’entrée et de sortie.

Ils ont également des propriétés qui permettent la manipulation de leur comportement.

Le contrôle de flux est appliqué aux files d’attente pour permettre différents taux de consommation entre les opérateurs.

La détection de Deadlock peut également être appliquée.

DATAFLOW ET PARALLÉLISME

Le modèle de flux de données fournit un pipeline dédié au parallélisme par sa nature même.

Les opérateurs normalement ne sont pas obligés de voir la totalité de leur entrée avant de produire la sortie.

Pendant que chaque opérateur travaille, un pipeline de données fait son chemin à travers le graphe. Chaque opérateur peut être représenté par un fil, ou peut-être un ensemble de fils.

Avec de nombreux opérateurs dans un graphe de flux de données fonctionnant en parallèle, le dataflow tire facilement parti des processeurs multicœurs.

PARTITIONNEMENT HORIZONTAL

Le dataflow prend également en charge le partitionnement horizontal, permettant aux données d'être segmentées et appliquées à une section répliquée d'un graphe.

Ce partitionnement peut être dynamique, en ajustant le nombre de ressources informatiques disponibles au moment de l'exécution.

Le partitionnement horizontal est un excellent moyen de « diviser pour régner » des problèmes où la dépendance de données n'est pas un problème.

Il peut à la fois s'appliquer à des processeurs multicœurs ou plusieurs nœuds dans un cluster.

ORIGINE DU PARADIGME

Une définition de flux de données fréquemment donnée dans les manuels de science informatique décrit une architecture selon laquelle la modification de la valeur d'un des résultats entraîne le recalcul automatique d'autres variables dépendantes.

Modèle ≪tableur≫

Le fondateur du paradigme dataflow, Gilles Kahn, définit le modèle de programmation Kahn Process Networks (KPNs).

Selon Wikipedia, KPN définit un modèle de programmation ≪ . . . où un groupe de processus séquentiels déterministes communiquent à travers des canaux FIFO non bornés. ≫

ORIGINE DU PARADIGME

Les KPNs ont été initialement conçus pour la programmation distribuée.

Ils sont également très utilisés pour les systèmes de modélisation de traitement du signal (opérateurs = filtres).

Le concept original de dataflow (feuilles de calcul) et KPNs ont, au fil du temps, évolué vers l'architecture logicielle de flux de données.

QUALITÉS DU DATAFLOW

Les concepts de flux de données sont faciles à saisir conduisant à « l'évolutivité de conception. »

Le dataflow se prête bien à des environnements graphiques, ce qui apporte au calcul haute performance un niveau de convivialité très accessible.

Le Dataflow fournit un moyen de composer des applications évolutives utilisant une approche modulaire. Il le fait tout en faisant abstraction des problèmes de bas niveau tels que les fils, la synchronisation de la mémoire, et les conditions de parcours.

Dataflow empreinte les bonnes qualités de la programmation fonctionnelle.

Par exemple, les files d'attente de flux de données sont immuables, ce qui élimine les préoccupations au sujet de la synchronisation de la mémoire ou des effets secondaires communs des opérateurs en amont.

L'immuabilité permet également aux files d'attente d'avoir plusieurs lecteurs pour une bonne réutilisation fonctionnelle.

Il n'y a pas d'effet de bord.

Il est facile de regarder un graphe de dataflow et deviner comment il fonctionne et ce qu'il fait.

Le modèle permet le chevauchement des opérations d'Entrées / Sorties avec calcul.

C'est une approche de la parallélisation totalement fonctionnelle

Cela répond à un problème majeur dans le traitement des « big data » pour les processeurs multi-coeurs de base d'aujourd'hui : alimenter suffisament rapidement les processeurs avec des données.

Le Dataflow passe facilement à l'échelle vers, contrairement à Hadoop ou Map Reduce, qui ne passent pas à l'échelle vers le bas en raison de leur complexité intrinsèque.

SOUTIEN AUX BIG DATA

L'architecture de flux de données exploite naturellement les processeurs multi-coeurs.

Les mêmes principes peuvent être appliqués à des grappes multi-noeuds par l'extension des files d'attente de flux de données sur les réseaux avec un graphe de dataflow exécuté sur plusieurs systèmes en parallèle.

Le modèle de la composition de la construction des graphes de flux de données permet la réplication de morceaux du graphe à travers plusieurs nœuds.

La portée du dataflow s'étend donc aux problèmes de big data.

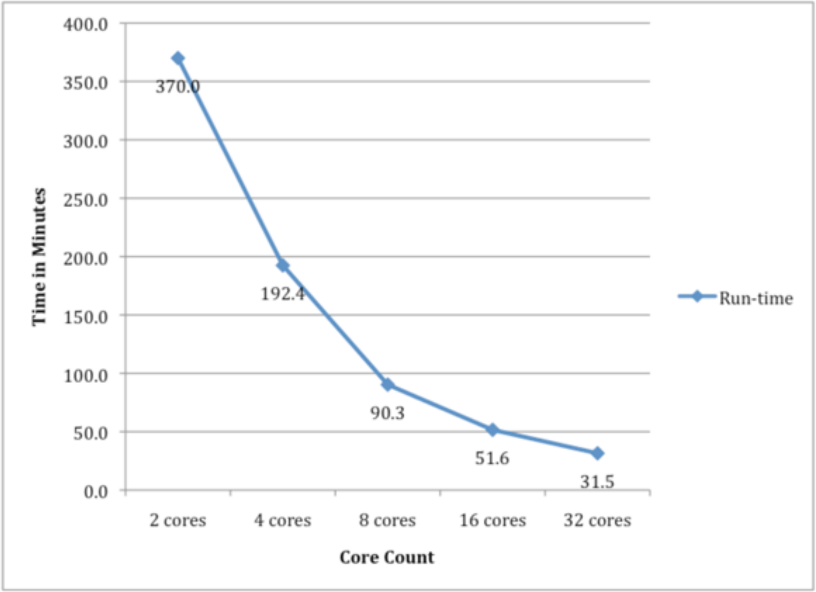
PERFORMANCE

Benchmark MalStone B10 : 10 milliards de lignes de données de log dans cinq domaines, pour mesurer la performance des frameworks de flux de données.

Presque un téraoctet de données.

Machine à nœud unique avec 32 cœurs.

Le code extrait l'année et la semaine à partir d'un champ timestamp et agrège un indice qui est calculé à partir de l'id du site, l'année et la semaine.


400.0
350.0
300.0
250.0
200.0
150.0
100.0
50.0
0.0
Time in Minutes
370.0
192.4
90.3
51.6
31.5
2 cores
4 cores
8 cores
16 cores
32 cores
Core Count
Run-time

De nombreux langages existent pour implémenter le modèle dataflow.

Exemple de framework récent, open-source et facile à intégrer dans Eclipse : Orcc

http ://orcc.sourceforge.net/

On peut également citer Oz, qui répond à de nombreux paradigmes, dont le dataflow.

Dans ce cours, nous allons approcher le paradigme dataflow en scheme à l'aide de la structure de données `stream`...

Les suites servent d'interfaces standard pour combiner les modules d'un programme.

On a vu des abstractions puissantes pour manipuler les suites, comme `map`, `filter` et `accumulate`. C'est une façon élégante de programmer.

Mais si on représente les suites comme des listes, cette élégance coûte cher en efficacité (en temps et en espace), car les programmes doivent construire et copier des structures de données potentiellement grandes, à chaque étape.

EXEMPLE D'INEFFICACITÉ DES LISTES POUR IMPLÉMENTER LES SUITES

On considère les deux programmes suivants pour calculer la somme des entiers premiers dans un intervalle d'entiers $[a, b]$:

```
(define (sum-primes a b)
  (define (iter count accum)
    (cond ((> count b) accum)
          ((prime? count) (iter (+ count 1) (+ count accum)))
          (else (iter (+ count 1) accum))))
  (iter a 0))

(define (sum-primes a b)
  (accumulate +
              0
              (filter prime? (enumerate-interval a b))))
```

EXEMPLE D'INEFFICACITÉ DES LISTES POUR IMPLÉMENTER LES SUITES

Le premier programme n'a besoin de stocker que la somme en train d'être calculée.

Dans le deuxième programme, le filtre ne peut pas faire de test tant que `enumerate-interval` n'a pas terminé de construire une liste complète des nombres dans l'intervalle. La fonction `filter` génère une autre liste, qui est à son tour traitée pour réaliser la somme.

Même problème si on veut le deuxième entier premier dans l'intervalle $[10000, 1000000]$.

```
(car (cdr (filter prime?
                  (enumerate-interval 10000 1000000))))
```

EXEMPLE D'INEFFICACITÉ DES LISTES POUR IMPLÉMENTER LES SUITES

Cette expression trouve le deuxième nombre premier, mais le temps est déraisonnable. On construit une liste de presque un million d'entiers, on la filtre en testant la primalité de chaque élément, puis on ignore presque l'intégralité du résultat.

Dans un style de programmation plus traditionnel, on intercalerait l'énumération et le filtrage, en s'arrêtant dès le deuxième nombre premier.

LA STRUCTURE DE DONNÉES `STREAM` (= FLOT)

Les flots permettent de manipuler des suites d'éléments sans le coût inhérent à leur implémentation sous forme de listes.

Le meilleur des deux mondes : programmation élégante et efficacité.

Principe : on construit le flot seulement de façon partielle, en fonction du besoin du programme qui le consomme.

INTERFACE DES STREAM

En surface, les flots sont justes des listes, avec des fonctions de manipulation ayant des noms différents.

```
cons-stream ;; constructeur
stream-car ;;accesseurs
stream-cdr
stream-null?
```

```scheme
(define (stream-ref s n)
  (if (= n 0)
      (stream-car s)
      (stream-ref (stream-cdr s) (- n 1))))

(define (stream-map proc s)
  (if (stream-null? s)
      the-empty-stream
      (cons-stream (proc (stream-car s))
                   (stream-map proc (stream-cdr s)))))

(define (stream-for-each proc s)
  (if (stream-null? s)
      'done
      (begin (proc (stream-car s))
             (stream-for-each proc (stream-cdr s)))))
```

Stream-for-each est utile pour visualiser les flots :

```
(define (display-stream s)
  (stream-for-each display-line s))

(define (display-line x)
  (newline)
  (display x))
```

L'implémentation est basée sur une forme spéciale appelée `delay`. L'évaluation de `(delay <exp>)` n'évalue pas l'expression `<exp>`, mais renvoie un objet « différé » qui promet d'évaluer l'expression plus tard.

La procédure `force` prend un objet différé et réalise l'évaluation.

```
(define (cons-stream a b) (cons a (delay b))

(define (stream-car stream) (car stream))

(define (stream-cdr stream) (force (cdr stream)))
```

RETOUR À L'EXEMPLE

```
(stream-car
 (stream-cdr
  (stream-filter prime?
                (stream-enumerate-interval 10000 1000000))))
```

On appelle `stream-enumerate-interval` sur les arguments 10,000 et 1,000,000.

```
(define (stream-enumerate-interval low high)
  (if (> low high)
      the-empty-stream
      (cons-stream
       low
       (stream-enumerate-interval (+ low 1) high))))
```

Le résultat renvoyé sur cet appel par `stream-enumerate-interval` est

```
(cons 10000
      (delay (stream-enumerate-interval 10001 1000000)))
```

```
(define (stream-filter pred stream)
  (cond ((stream-null? stream) the-empty-stream)
        ((pred (stream-car stream))
         (cons-stream (stream-car stream)
                      (stream-filter pred
                                     (stream-cdr stream))))
        (else (stream-filter pred (stream-cdr stream)))))
```

La procédure `stream-filter` teste le `stream-car` du flot qui est 10,000.

RETOUR À L'EXEMPLE

Comme 10000 n'est pas premier, `stream-filter` examine le `stream-cdr` de son flot d'entrée. Cela force l'évaluation du `stream-enumerate-interval` différé, qui renvoie :

```
(cons 10001
      (delay (stream-enumerate-interval 10002 1000000)))
```

Et ainsi de suite jusque 10007 qui est premier.

RETOUR À L'EXEMPLE

`stream-filter` renvoie alors

```
(cons-stream (stream-car stream)
             (stream-filter pred (stream-cdr stream)))
```

C'est-à-dire

```
(cons 10007
      (delay
        (stream-filter
         prime?
          (cons 10008
                (delay
                  (stream-enumerate-interval 10009
                                             1000000))))))
```

Ce résultat est passé à `stream-cdr` dans l'expression originelle. Cela force le `stream-filter` différé, qui à son tour force le `stream-enumerate-interval` différé jusqu'à ce qu'il trouve le nombre premier suivant, 10009. Finalement, le résultat passé au `stream-car` dans l'expression originelle est :

```
(cons 10009
      (delay
        (stream-filter
         prime?
         (cons 10010
               (delay
                 (stream-enumerate-interval 10011
                                            1000000))))))
```

RETOUR À L’EXEMPLE

`stream-car` renvoie 10009 et le calcul est terminé. On n’a testé qu’autant d’entiers qu’il fallait, et l’intervalle n’a pas été énuméré plus loin.

IMPLÉMENTATION DE DELAY ET FORCE

```
(delay <exp>)
```

est un sucre syntactique pour

```
(lambda () <exp>)
```

`force` appelle simplement la procédure sans argument produite par `delay`

```
(define (force delayed-object)
  (delayed-object))
```

Une séance CAML