Université Lyon 1 – Licence Sciences et Technologies

UE LIF 3

Algorithmique et Programmation Fonctionnelle et Récursive

Responsables : Florence Zara, Nathalie Guin[1]

Support de cours

Avertissement

Ce support de cours est constitué d'une copie des transparents utilisés en cours. Il ne s'agit pas d'un polycopié. Il est mis à votre disposition afin de vous éviter d'avoir à les recopier. Cela ne vous dispense pas pour autant de prendre des notes sur tout ce que est dit autour des transparents, et encore moins d'assister au cours. Votre attention est attirée sur le fait que les épreuves de contrôle des connaissances de cette UE portent sur l'ensemble de ce qui vous a été dit en CM, TD et TP, et pas uniquement sur ce qui est écrit dans ces transparents.

Page web de l'UE LIF3 : https://liris.cnrs.fr/~fzara/LIF3/

Vous trouverez sur cette page :

- le planning CM-TD-TP de l'UE,
- les horaires et les salles des TD-TP,
- les transparents des cours consultables en ligne,
- les sujets de TD et TP,
- un lien vers le site web de DrScheme, à partir duquel vous pouvez télécharger le logiciel.

[1] Les transparents de cours ont été réalisés par N. Guin

LIF3

Plan Licence

Présentation de l'UE

Modalité de Contrôle des Connaissances

Objectifs pédagogiques

Plan Licence

- Toutes les UEs du L1 et L2 en Contrôle Continu Intégral
  - plusieurs notes pour vous évaluer
  - plus de notion d'examen et de seconde session d'examen
- Contrôle terminal le **mardi 10 mai 2016** de 9h45 à 11h15
- Séances de soutien
  - sur avis des intervenants de TD
  - 2 ou 3 séances sur le semestre
- Harmonisation des notes en fin de semestre entre les groupes

Présentation de l'UE

- Responsables de l'UE
  - Florence Zara – bâtiment Nautibus
  - Nathalie Guin – bâtiment Nautibus
  - Marie Lefevre – bâtiment Nautibus
- Site Web de l'UE
  - https://liris.cnrs.fr/~fzara/LIF3
  - Planning (salles, horaires)
  - Répartition des groupes de TDs et TPs
  - Supports des CMs, TDs et TPs
  - Corrigés des TPs
  - etc.

Planning de l'UE

- Planning détaillé sur la page Web de LIF3
  - Début des TDs : semaine du 25 janvier 2016
  - Début des TPs : semaine du 1 février 2016
- Attention :
  - **Jeudi 21/01 :** cours de 14h à 15h30 et cours de 15h45 à 17h15
  - **Mardi 26/01** : TD de 9h45 à 11h15 et TD de 11h30 à 13h
- Les créneaux de soutien mis sur le planning sont destinés aux étudiants convoqués

Modalité de Contrôle des Connaissances

- Cette UE est en Contrôle Continu Intégral
  - **il n'y aura donc pas d'examen écrit pendant la « session d'examens », ni de seconde session**
- Le contrôle continu est constitué de plusieurs épreuves :
  - **interrogations écrites en début de TD** (coefficient 0,20)
  - TP noté en conditions d'examen (TP7, coefficient 0,25) : **mardi 5 avril 2016**
  - TP projet (TP 8 et 9, coefficient 0,15) : évaluation le **mardi 3 mai 2016 en salle de TP**
  - épreuve écrite en amphi (coefficient 0,4) : **mardi 10 mai 2016 de 9h45 à 11h15**

Objectifs pédagogiques de l'UE (1)

- **Cours 1 : Algorithmes, début en Scheme**
  - Algorithmes itératif *vs.* récursif
  - Syntaxe du langage de programmation Scheme
- **Cours 2 : Complexité, listes**
  - Complexité en temps et complexité en espace
  - Utilisation des listes en Scheme
- **Cours 3 : Let**
  - Mémorisation
- **Cours 4 : Tris**
  - Tris par insertion et par fusion

Objectifs pédagogiques de l'UE (2)

- **Cours 5 : Récursivité profonde**
  - Plusieurs niveaux de récursivité
- **Cours 6 : Arbres**
  - Utilisation des arbres en Scheme
- **Cours 7 : Programmation d'ordre supérieur**
  - Fonctions en arguments, fonctions en résultats
  - Abstraction
- **Cours 8 : Logique**
  - Logique des propositions
  - Algèbre de Boole

Algorithmique et programmation fonctionnelle et récursive

Algorithme itératif / récursif

Programmation impérative / fonctionnelle

Qu'est-ce qu'un algorithme ?

On souhaite résoudre le problème :

Trouver le minimum d'une liste de nombres :
par exemple trouver le minimum de (3 6,5 12 -2 0 7)

➡ Expliquer à une machine comment trouver le minimum **de n'importe quelle** liste de nombres

➤ Langage commun entre la machine et nous : Scheme

Définition d'un algorithme

- Un algorithme est une méthode
  - Suffisamment **générale** pour pouvoir traiter toute une classe de problèmes
  - Combinant des **opérations** suffisamment **simples** pour être effectuées par une machine

Rappel : la machine n'est pas intelligente

- L'exécution d'un algorithme ne doit normalement pas impliquer des décisions subjectives, ni faire appel à l'utilisation de l'intuition ou de la créativité
- Exemple : une recette de cuisine n'est pas un algorithme si elle contient des notions vagues comme « ajouter de la farine »
- La machine fait ce qu'on lui demande de faire

Mon premier algorithme

- Déterminer le minimum d'une liste de nombres
  - par exemple trouver le minimum de la liste
    (3 6,5 12 -2 0 7)

Méthode itérative

- Je parcours la liste de gauche à droite, en retenant à chaque pas le minimum « pour l'instant »

  (3 6,5 12 -2 0 7)

  - Entre 3 et 6,5, c'est 3
  - Entre 3 et 12, c'est 3
  - Entre 3 et -2, c'est -2
  - Entre -2 et 0, c'est -2
  - etc.

De quoi ai-je besoin pour écrire l'algorithme ? (1)

- La séquence

  Début
  Instruction(s)
  Fin

- L'affectation

  Variable ← Expression

  Exemple :
  - A ← 2
  - B ← A+2*racine(15)

De quoi ai-je besoin pour écrire l'algorithme ? (2)

- La conditionnelle (1)

  Si Condition Alors
  Instruction(s)
  FinSi

- Exemple :

  Si (A > 27) Alors
  B ← A*3
  FinSi

De quoi ai-je besoin pour écrire l'algorithme ? (3)

- La conditionnelle (2)

  Si Condition Alors
  Instruction(s)
  Sinon
  Instruction(s)
  FinSi

- Exemple :

  Si ((A<10) et
  (B > racine(A*5))) Alors
  B ← A*3
  A ← A+B
  Sinon
  A ← A+2
  B ← A*B
  FinSi

De quoi ai-je besoin pour écrire l'algorithme ? (4)

La **condition** est :

- soit une condition élémentaire
- soit une condition complexe, c'est à dire une conjonction, disjonction, ou négation de conditions élémentaires et/ou complexes

De quoi ai-je besoin pour écrire l'algorithme de manière itérative ? (5)

- L'itérative, ou boucle

  TantQue Condition Faire
  Instruction(s)
  FinTantQue

- Exemple :

  TantQue i<10 Faire
  a ← a*i
  i ← i+1
  FinTantQue

Opérateurs booléens

- Pour écrire une conjonction, disjonction, ou négation de conditions, on a besoin des opérateurs booléens ET, OU, NON
- Une variable booléenne est une variable dont les valeurs peuvent être vrai ou faux
- Un opérateur booléen est un opérateur dont les arguments sont des variables booléennes et dont le résultat est booléen

L'opérateur ET

X	Y	X ET Y
Vrai	Vrai	Vrai
Vrai	Faux	Faux
Faux	Vrai	Faux
Faux	Faux	Faux

L'opérateur OU

X	Y	X OU Y
Vrai	Vrai	Vrai
Vrai	Faux	Vrai
Faux	Vrai	Vrai
Faux	Faux	Faux

L'opérateur NON

X	NON X
Vrai	Faux
Faux	Vrai

Algorithme itératif

Soient - L la liste
- premier, longueur, ième et écrire des primitives (*i.e.* algorithmes déjà définis)

```
Définition de minimum(L)
Début
    min ← premier(L)
    i ← 2
    TantQue i <= longueur(L) Faire
        Si ième(L,i) < min Alors
            min ← ième(L,i)
        FinSi
        i ← i+1
    FinTantQue
    Écrire(« Le minimum est »)
    Écrire(min)
Fin
```

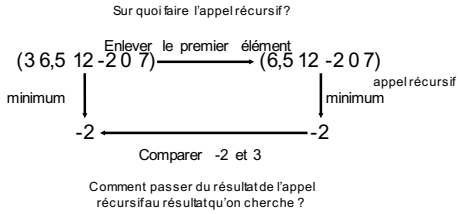

Méthode récursive (1)

- Pour trouver le minimum de la liste (3 6,5 12 -2 0 7)
- On suppose le problème résolu pour la liste privée de son premier élément *i.e.* (6,5 12 -2 0 7)
- Soit min le minimum de cette sous-liste, ici -2
- Le minimum de la liste complète s'obtient par comparaison entre le premier élément de la liste (ici 3) et min (ici -2)

Méthode récursive (2)

- Comment résout-on le problème pour la sous-liste ?
  En faisant le même raisonnement.
  C'est la **récursivité**.
- Quand s'arrête-t-on ?
  Quand on ne peut plus parler de sous-liste, *i.e.* quand la liste a un seul élément. C'est alors le minimum.

Illustration de la méthode



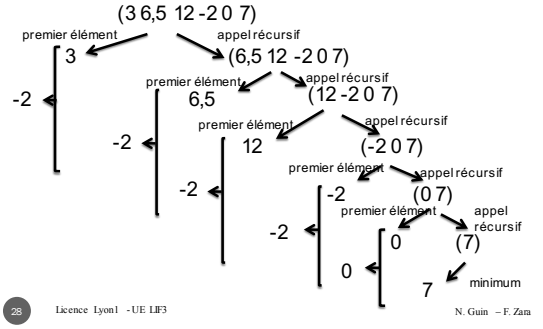
Algorithme récursif

Soient vide?, reste et premier des fonctions primitives

```
Définition de la fonction minimum(L)

Si vide?(reste(L)) Alors
        retourne premier(L)
Sinon
        Si premier(L) < minimum(reste(L)) Alors
                retourne premier(L)
        Sinon
                retourne minimum(reste(L))
        FinSi
FinSi
```


Illustration de l'algorithme



Remarques

- Minimum est ici une fonction, le mot retourne permet de dire quel est son résultat
- Minimum est l'appel récursif
- En programmation fonctionnelle, on n'a plus besoin de la séquence
- En programmation récursive, on n'a plus besoin de la boucle

Pour écrire un algorithme récursif

- Il faut choisir
  - Sur quoi on fait l'appel récursif
  - Comment on passe du résultat de l'appel récursif au résultat que l'on cherche
  - Le cas d'arrêt

Structure type d'une fonction récursive

Si	je suis dans le cas d'arrêt
Alors	voila le résultat
Sinon	le résultat est le résultat de l'application d'une fonction sur le résultat de l'appel récursif

Les bugs

- Mon algorithme est faux car son résultat n'est pas défini si la liste est vide ou si elle contient autre chose que des nombres
- Pour éviter les bugs il faut :
  - Définir le problème aussi bien que possible
    C'est la spécification
  - Tenter de prouver que son programme répond à la spécification
  - Passer des jeux d'essai, aussi divers et tordus que possible

Pour résumer

LIF1 : Programmation	Impérative	Itérative
Langage C	Séquence (faire les choses l'une après l'autre)	Boucle (répéter)
⇩	⇩	⇩
LIF3 : Programmation	Fonctionnelle	Récursive
Langage Scheme	Appliquer une fonction à des arguments pour obtenir un résultat Composer les fonctions pour enchaîner les traitements	La répétition est assurée par l'enchaînement des appels récursifs Le test de la boucle est remplacé par le cas d'arrêt

Débuts en Scheme

Évaluer une expression

Définir une fonction

Notion de fonction

- Une fonction prend des arguments et retourne **un** résultat
- Arguments et résultats peuvent être de n'importe quel type :
  - Nombre
  - Booléen
  - Caractère
  - Chaîne de caractères
  - Liste
  - Fonction

Écriture de l'appel à une fonction (1)

- Syntaxe :
  - Parenthèse ouvrante
  - Nom de la fonction
  - Espace
  - Premier argument
  - Espace
  - Deuxième argument
  - Etc
  - Parenthèse fermante

(NomFct Arg1 Arg2 … Argn)

Écriture de l'appel à une fonction (2)

- Sémantique : il faut donner à la fonction le bon nombre d'arguments, et du bon type
- Exemples :
  - (+ 5 13) retourne 18
  - (- 10 b) retourne la différence si *b* a une valeur numérique, une erreur sinon
  - (+ (* 2 5) (- 3 1)) retourne 12
  - (* 5) n'est pas correct
  - (/ 5 "a") non plus

Évaluation de l'appel à une fonction

- Lorsqu'on lui fournit un appel de fonction, Scheme
  - Évalue chacun des arguments
  - Regarde s'il connaît la fonction, sinon affiche un message d'erreur
  - Applique la fonction aux résultats de l'évaluation des arguments
  - Affiche le résultat
- C'est un processus récursif

Exemples d'erreurs

- (1 + 2) → erreur : 1 n'est pas une fonction
- (toto (1 2 3)) → erreur : 1 n'est pas une fonction
- Dans certains cas particuliers, les arguments ne sont pas évalués avant l'application de la fonction. On parle alors de forme spéciale plutôt que de fonction

Les variables

- Dans le langage Scheme, une variable se nomme symbole
- L'affectation revient à attribuer une valeur à un symbole. On utilise pour cela la forme spéciale define
- Exemples :
  (define a 5)
  (define b 2)
  (+ a b) → 7

La forme spéciale quote

- La forme spéciale quote permet d'empêcher l'évaluation
- Exemples :
  (define a 5)
  a → 5

  (quote a) → a
  (quote (+ 1 2)) → (+ 1 2)
  '(+ 1 2) → (+ 1 2)

La forme spéciale eval

- À l'inverse de quote, eval force l'évaluation
- Exemples :
  (eval '(+ 3 2)) → 5

  (define toto 5)
  (define tata toto)
  tata → 5

  (define titi 'toto)
  titi → toto
  (eval titi) → 5

Définition d'une fonction

- Syntaxe :

```
(define fonction
  (lambda liste-des-arguments
    corps-de-la-fonction))
```

- Exemple :

```
(define plus-1
    (lambda (x)
        (+ x 1)))
```

- Test : (plus-1 3) → 4

Spécification d'une fonction

```
; description de ce que fait la fonction
(define fonction ; -> type du résultat
    (lambda liste-des-arguments  ; type des arguments
        corps-de-la-fonction))
```

- Exemple :

```
; ajoute 1 à un nombre
(define plus-1 ; -> un nombre
        (lambda (x) ; x un nombre
                (+ x 1)))
```


L'alternative

- L'alternative est définie en Scheme par la forme spéciale if
- Syntaxe :
  (if condition valeur-si-vrai valeur-si-faux)
- Exemples :
  (if (> 3 2) 'yes 'no) → yes
  (if (> 2 3) 'yes 'no) → no
  (if (> 3 2) (- 3 2) (+ 3 2)) → 1

Quelques fonctions prédéfinies (1)

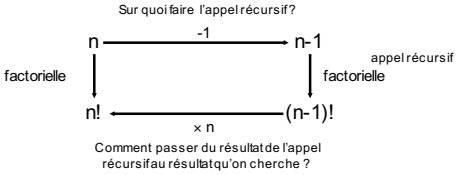
- Opérateurs arithmétiques :
  +, -, *, /, sqrt, min, max, abs, etc.
  (sqrt 9) → 3
  (min 5 2 1 3) → 1
- Opérateurs booléens : not, or, and
  (not #t) → #f
  (and (> 3 2) (> 2 5)) → #f
  (or (> 3 2) (> 2 5)) → #t

Quelques fonctions prédéfinies (2)

- Opérateurs de comparaison :
  - =, <, >, <=, >= pour les nombres
  - eq? pour tout sauf les listes et les chaînes de caractères
  - equal? pour tout y compris les listes
- Pour tester le type d'un objet :
  boolean?, number?, symbol?, string?

Ma première fonction récursive : choix de la méthode

- On veut écrire une fonction qui étant donné un entier n calcule n!



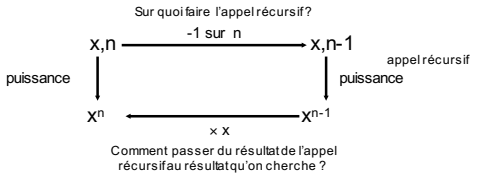
Ma première fonction récursive : écriture

- Cas d'arrêt : 0! = 1
- Récursivité : n! = 1 x 2 x 3 x … x n = n x (n-1)!

```
(define factorielle ; → entier positif
  (lambda (n) ; n entier positif
      (if     (= n 0)
              1
              (* n (factorielle (- n 1))))))
```


Une autre fonction récursive : choix de la méthode

- On veut écrire une fonction qui étant donné un nombre x et un entier positif n calcule x^n



Une autre fonction récursive : écriture

- Cas d'arrêt : $x^0 = 1$
- Récursivité : $x^n = x \times x \times x \times \ldots \times x = x \times x^{(n-1)}$

```
(define puissance ; → nombre
  (lambda (x n) ; x nombre, n entier positif
      (if     (= n 0)
              1
              (* x (puissance x (- n 1))))))
```

Complexité des algorithmes

Complexité en temps

Complexité en espace

Complexité d'un algorithme

- Il faut :
  - que la machine trouve le plus vite possible
    - Complexité en temps
  - qu'elle trouve en utilisant aussi peu de place mémoire que possible
    - Complexité en espace

Complexité en temps de l'algorithme itératif du minimum

```
Définition de minimum(L)                    Soit n la longueur de la liste L
Début
    min ← premier(L)                        1 affectation (initialisation)
    i ← 2                                   1 affectation (initialisation)
    TantQue i <= longueur(L) Faire          n comparaisons
        Si ième(L,i) < min Alors            n-1 comparaisons
            min ← ième(L,i)                 m affectations
        FinSi
            i ← i+1                         n-1 affectation (incrémentation)
    FinTantQue
    Écrire(« Le minimum est »)              1 écriture
    Écrire(min)                             1 écriture
Fin
```


Le nombre m dépend de la liste

- Meilleur cas : m=0 si le premier nombre de la liste est le minimum
- Pire cas : m=n-1 si les nombres de la liste sont rangés en ordre décroissant
- Cas moyen : m=(n-1)/2 s'ils respectent une distribution parfaitement aléatoire

Complexité en espace de l'algorithme

- Un mot pour stocker le « minimum pour l'instant » (min)
- Un mot pour savoir où on en est dans la liste (i)

Complexité en temps de l'algorithme récursif du minimum

```
Définition de la fonction minimum(L)
Si vide?(reste(L)) Alors                    1 test
    retourne premier(L)
Sinon
    Si premier(L) < minimum(reste(L)) Alors 1 comparaison
            retourne premier(L)               + le nombre de
    Sinon                                     comparaisons de
            retourne minimum(reste(L))        l'appel récursif
    FinSi
FinSi
```

Complexité en temps de l'algorithme (2)

- Si n est la longueur de la liste
- Si C(i) est le nombre de comparaisons de l'algorithme pour une liste de longueur i
- Alors C(n) = 1+C(n-1)
  = 1+1+C(n-2)
  = …
  = 1+1+…+C(1)
  = 1+1+…+0
  = n-1

Résumé sur la complexité

- Choisir ce que l'on va compter
  - unité de comparaison des algorithmes
  - par exemple le nombre de comparaisons
- Ce qui importe est l'ordre de grandeur de la complexité
  - constant, log n, linéaire, n*log n, n^2, 2^n
- En LIF3 on s'intéressera essentiellement au nombre de fois où l'on parcourt une structure de donnée (liste, arbre)

Les listes en Scheme

car, cdr, cons
cond
list, append

Définition

- Une **liste** est une suite d'éléments rangés dans un certain ordre

  '(alpha 3 beta "delta" gamma)

Listes et paires en Scheme

- Une **paire** est une structure d'enregistrement avec deux champs appelés car et cdr
- Les champs car et cdr sont accédés par les fonctions car et cdr
- Une liste peut être définie récursivement comme :
  - soit une liste vide
  - soit une paire dont le cdr est une liste

Écriture d'une liste

- Les éléments d'une liste sont simplement compris entre parenthèses et séparés par des espaces
- La liste vide est écrite ()
- (a b c d e) et (a . (b . (c . (d . (e . ())))))
  sont deux notations équivalentes pour une liste de symboles

La fonction pair?

- (pair? x) retourne
  - #t si x est une paire,
  - #f sinon
- Exemples :
  - (pair? '(a . b)) → #t
  - (pair? '(a b c)) → #t
  - (pair? '()) → #f

Les fonctions car et cdr

- (car x) retourne le champ car de x
  - (car '(a b c)) → a
  - (car '((a) b c d)) → (a)
  - (car '(1 . 2)) → 1
  - (car '()) → error
- (cdr x) retourne le champ cdr de x
  - (cdr '((a) b c d)) → (b c d)
  - (cdr '(1 . 2)) → 2
  - (cdr '(a)) → ()
  - (cdr '()) → error
- Si x est une liste, la fonction car retourne le premier élément de la liste, et la fonction cdr retourne le reste de la liste

La fonction cons

- (cons x y) retourne une nouvelle paire dont le car est x et le cdr est y
  (si y est une liste, elle met x au début de y)
- Exemples :
  - (cons 'a '()) → (a)
  - (cons '(a) '(b c d)) → ((a) b c d)
  - (cons "a" '(b c)) → ("a" b c)
  - (cons 'a 3) → (a . 3)
  - (cons '(a b) 'c) → ((a b) . c)
- Pour que le résultat du cons soit une liste, il faut donc que le deuxième argument soit une liste

Fonctions de test

- (list? x) retourne #t si x est une liste, #f sinon
  - (list? '(a b c)) → #t
  - (list? '()) → #t
  - (list? '(a . b)) → #f
- (null? x) retourne #t si x est la liste vide, #f sinon

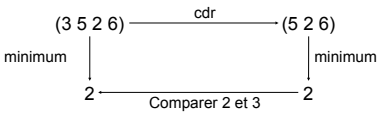
Implémentation de l'algorithme récursif du minimum

Définition de la fonction minimum(L)

```
Si vide?(reste(L)) Alors
    retourne premier(L)
Sinon
    Si premier(L) < minimum(reste(L)) Alors
            retourne premier(L)
    Sinon
            retourne minimum(reste(L))
    FinSi
FinSi
```

```
(define minimum ; → nombre
   (lambda (l) ; l liste de nombres non vide
         (if (null? (cdr l))
            (car l)
            (if  (<   (car l)
                      (minimum (cdr l)))
               (car l)
               (minimum (cdr l))))))
```

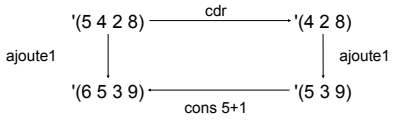

Illustration de la méthode



Une fonction qui retourne une liste

- Une fonction qui ajoute 1 à tous les éléments d'une liste de nombres

(ajoute1 '(5 4 2 8)) → (6 5 3 9)



Écriture de la fonction

```
(define ajoute1; → liste de nombres
  (lambda (l) ; l liste de nombres
      (if  (null? l)
            '()
            (cons (+ (car l) 1)
                  (ajoute1 (cdr l))))))
```


Relâcher des préconditions

Une fonction qui ajoute 1 à tous les nombres d'une liste

```
(define ajoute1; → liste
  (lambda (l) ; l liste
      (if  (null? l)
            '()
            (if (number? (car l))
                  (cons    (+ (car l) 1)
                           (ajoute1 (cdr l)))
                  (cons (car l)
                           (ajoute1 (cdr l)))))))
```

(ajoute1 '(2 "a" 1 (toto) 5)) → (3 "a" 2 (toto) 6)

La forme spéciale cond

- La forme spéciale cond permet d'écrire plus simplement des if imbriqués

```
(cond
      (test1 valeur1)
      (test2 valeur2)
      …
      (else valeurN))
```


Application à ajoute1

```
(define ajoute1; → liste
  (lambda (l) ; l liste
    (cond
      ((null? l) '())
      ((number? (car l)) (cons (+ (car l) 1)
                               (ajoute1 (cdr l))))
      (else (cons (car l) (ajoute1 (cdr l)))))))
```


Mettre en facteur dans un if

```
(if (number? (car l))
   (cons (+ (car l) 1)
         (ajoute1 (cdr l)))
   (cons (car l)
         (ajoute1 (cdr l))))
```

```
(cons
   (if (number? (car l))
         (+ (car l) 1)
         (car l))
   (ajoute1 (cdr l)))
```

Ces deux expressions sont équivalentes

La fonction list

- La fonction list est une fonction qui permet de construire une liste, comme la fonction cons
- Elle prend un nombre quelconque d'arguments et les met tous dans une liste

(list 'a (+ 3 2) "toto" 'b) → (a 5 "toto" b)

Ne pas confondre list et list?

La fonction append

- La fonction append permet de concaténer des listes
- Elle prend un nombre quelconque d'arguments

(append '(a (b c) d) '(e f)) → (a (b c) d e f)

cons, list et append : les différences

Ces trois fonctions permettent toutes de construire des listes, mais ont chacune un comportement différent

- cons prend deux arguments, le deuxième étant obligatoirement une liste
- list prend un nombre quelconque d'arguments de types quelconques
- append prend un nombre quelconque d'arguments qui doivent tous être des listes

cons, list et append : exemples

- (cons '(a b) '(c d)) → ((a b) c d)
- (list '(a b) '(c d)) → ((a b) (c d))
- (append '(a b) '(c d)) → (a b c d)

Raccourcis pour composer les fonctions car et cdr

- Au lieu d'écrire (car (cdr L)), on peut écrire (cadr L)
- Au lieu d'écrire (cdr (cdr (cdr L))), on peut écrire (cdddr L)
- Et ainsi de suite (au maximum 4 caractères entre le c et le r)

Mémorisation

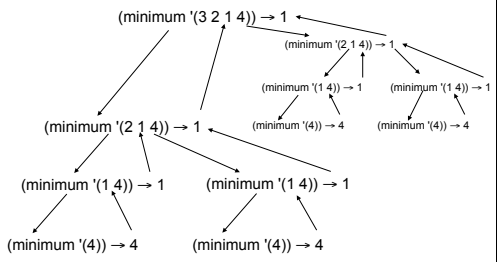
let

Mémoriser : pour quoi faire ?

Reprenons notre programme minimum :

```
(define minimum ; → nombre
  (lambda (l) ; l liste de nombres non vide
      (if (null? (cdr l))
        (car l)
        (if  (< (car l) (minimum (cdr l)))
          (car l)
          (minimum (cdr l))))))
```


Mémoriser : pour quoi faire ? (2)



Comment mémoriser ?

- On souhaite conserver le résultat du premier appel à minimum pour s'en resservir au lieu de provoquer le deuxième appel
- On définit donc un identificateur local (variable locale) grâce à un let

Syntaxe du let

(let ((ident$_1$ val$_1$)
(ident$_2$ val$_2$)
…
(ident$_N$ val$_N$))
corps)

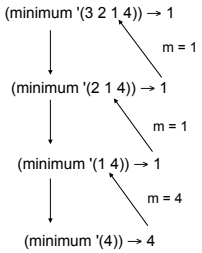
Fonctionnement du let

- Les val$_i$ sont évalués (dans un ordre quelconque) et ces valeurs sont affectées aux ident$_i$
- Dans le corps, on peut utiliser les ident$_i$
- Attention : les ident$_i$ ne sont pas définis à l'extérieur du corps

Application au programme minimum

```
(define minimum ; → nombre
  (lambda (l) ; l liste de nombres non vide
      (if     (null? (cdr l))
              (car l)
              (let ((m (minimum (cdr l))))
                    (if     (< (car l) m)
                            (car l)
                            m)))))
```


Fonctionnement du nouveau programme



Autre exemple

- Écrire une fonction qui calcule

$$\frac{3\sqrt{\frac{x^2}{2}+1}}{\sqrt{\frac{x^2}{2}}}$$

```
(define calcule ; → nombre
  (lambda (x) ; x nombre non nul
      (/ (+ (* 3 (sqrt (/ (sqr x) 2))) 1)
        (sqrt (/ (sqr x) 2))))))
```


Amélioration

- (define calcule ; → nombre
  (lambda (x) ; x nombre strictt positif
    (let ((c (sqrt (/ (sqr x) 2))))
      (/ (+ (* 3 c) 1) c))))
- L'utilisation du let permet ici une simplification d'écriture, mais n'améliore pas significativement la complexité de l'algorithme
- Dans le cas d'un appel récursif comme dans le programme minimum, l'utilisation du let est primordiale pour la complexité

Quand les identificateurs sont liés

- (define toto ; → nombre
  (lambda (x) ; x nombre
    (let ((a (sqr x))
            (b (+ (* 2 a) 1)))
      (if   (< a 80)
            (* 3 (+ a 1))
            (sqrt b)))))
- → erreur car les affectations de a et b ont lieu dans un ordre quelconque

let*

- (define toto ; → nombre
  (lambda (x) ; x nombre
    (let* ((a (sqr x))
           (b (+ (* 2 a) 1)))
      (if   (< a 80)
            (* 3 (+ a 1))
            (sqrt b)))))
- Les évaluations des identificateurs se font séquentiellement dans un let*

Tris

Tri par insertion

Tri par fusion

Quel est le problème à résoudre ?

- Soit une liste de nombres : '(5 2 14 1 6)
- On souhaite la trier : '(1 2 5 6 14)

Algorithmes de tri

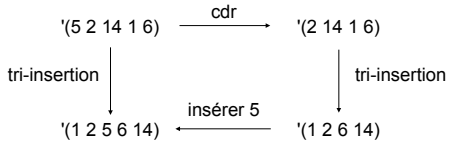
- Tris par sélection du minimum
  - tri-minimum (TP)
  - tri-bulles (TD)
- Tri par insertion
- Tri par fusion
- Tri rapide
- Tri par tas

Principes des tris par sélection

- On cherche le minimum de la liste, puis on recommence avec le reste de la liste
- Tri du minimum
  - fonction minimum
  - fonction enlève
- Tri bulles
  - fonction bulle, qui sélectionne le minimum et l'enlève de la liste en un seul passage

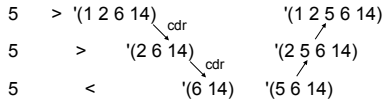
Tri par insertion : la méthode

- Principe : on trie récursivement le cdr de la liste, puis on y insère le car
- Exemple :



Insertion dans une liste triée : la méthode

- Principe : on compare l'élément à insérer avec le car de la liste
- Exemple : insérer 5 dans '(1 2 6 14)



Insertion dans une liste triée : la fonction

```
(define insere ; → liste de nombres triée
  (lambda (n l) ; n nombre, l liste de nombres triée
      (if    (null? l)
             (list n)
             (if    (< n (car l))
                    (cons n l)
                    (cons (car l) (insere n (cdr l)))))))
```


Tri par insertion : la fonction

```
(define tri-insertion ; → liste de nombres triée
  (lambda (l) ; l liste de nombres non vide
   (if (null? (cdr l))
    l
    (insere (car l) (tri-insertion (cdr l))))))
```


Tri par fusion : l'approche « Diviser pour régner »

- Structure récursive : pour résoudre un problème donné, l'algorithme s'appelle lui-même récursivement une ou plusieurs fois sur des sous problèmes très similaires
- Le paradigme « diviser pour régner » donne lieu à trois étapes à chaque niveau de récursivité : diviser, régner, combiner

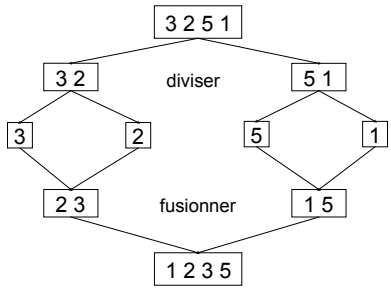
Diviser pour régner : 3 étapes

- Diviser le problème en un certain nombre de sous-problèmes
- Régner sur les sous-problèmes en les résolvant récursivement
  Si la taille d'un sous-problème est assez réduite, on peut le résoudre directement
- Combiner les solutions des sous-problèmes en une solution complète pour le problème initial

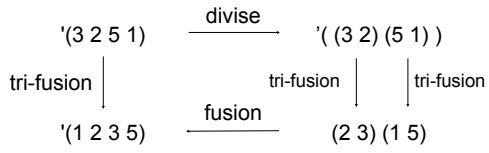
Tri par fusion : le principe

- Diviser : diviser la liste de n éléments à trier en deux sous-listes de n/2 éléments
- Régner : trier les deux sous-listes récursivement à l'aide du tri par fusion
- Combiner : fusionner les deux sous-listes triées pour produire la réponse triée

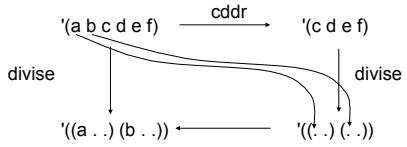
Un exemple



Tri par fusion : la méthode



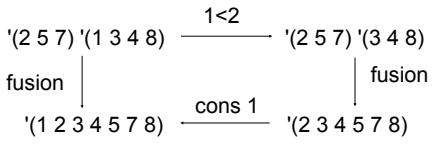
Diviser la liste en deux sous-listes : la méthode



Diviser la liste en deux sous-listes : la fonction

```
(define divise ; → liste de deux listes
 (lambda (l) ; l liste
  (cond       ((null? l) '(() ()))
      ((null? (cdr l)) (list l '()))
      (else (let ((r (divise (cddr l))))
              (list (cons (car l) (car r))
                    (cons (cadr l) (cadr r))))))))
```


Fusionner deux listes triées : la méthode



Fusionner deux listes triées : la fonction

```
(define fusion ; → liste de nb triée
 (lambda (l1 l2) ; listes de nb triées
  (cond ((null? l1) l2)
        ((null? l2) l1)
        ((< (car l1) (car l2))
              (cons (car l1) (fusion (cdr l1) l2)))
        (else
              (cons (car l2) (fusion l1 (cdr l2)))))))
```


Tri par fusion : la fonction

```
(define tri-fusion ; → liste de nb triée
 (lambda (l) ; liste de nb non vide
  (if (null? (cdr l))
     l
     (let ((r (divise l)))
       (fusion   (tri-fusion (car r))
                 (tri-fusion (cadr r)))))))
```

Calculs en remontant ou en descendant

- Jusqu'à présent, nous avons toujours effectué les calculs en remontant des appels récursifs
- Exemple : retour sur la fonction factorielle

```
(define factorielle ; → entier positif
  (lambda (n) ; n entier positif
    (if     (= n 0)
            1
            (* n (factorielle (- n 1))))))
```


Introduire un paramètre supplémentaire pour effectuer les calculs en descendant

```
(define factorielle-compteur; → entier positif
   (lambda (n) ; n entier positif
      (fact n 1)))

; effectue le calcul de factorielle(n) en utilisant un paramètre supplémentaire res
   dans lequel on effectue le calcul
(define fact ; → entier positif
 (lambda (n res) ; entiers positifs
  (if (= n 0)
     res
     (fact (- n 1) (* res n)))))
```


Remarques

- La fonction factorielle-compteur est celle qui répond à la spécification. Il est indispensable d'écrire une fonction qui répond à la spécification, même si elle ne fait rien d'autre que d'appeler la fonction fact. L'utilisateur n'a pas à savoir que nous utilisons un deuxième argument.
- La fonction fact est celle qui fait effectivement tout le travail.

Quel intérêt ?

- Dans la fonction fact, on a une récursivité terminale. Avec certains langages de programmation et certains compilateurs, cette récursivité terminale est « dérécursifiée » afin d'améliorer l'efficacité du programme.
- On se rapproche en effet d'une solution itérative :

```
res ← 1
TantQue n>0 Faire
   res ← res*n
   n ← n-1
FinTantQue
Afficher res
```

Récursivité profonde

Définition

- Une fonction va parcourir récursivement **en profondeur** une liste L si elle s'applique pour chaque liste l de cette liste L, de la même manière qu'elle s'applique sur L, et ceci de manière récursive : elle s'applique aussi sur les listes de l …
- On a ainsi deux niveaux de récursivité :
  - le premier, traditionnel, sur la structure de L,
  - et le second sur les éléments de L qui sont des listes

Premier exemple : la fonction somme

- Définissons la fonction somme qui additionne tous les nombres d'une liste quelconque

```
(define somme ; → nombre
  (lambda (L) ; L liste
    (cond ((null? L) 0)
          ((number? (car L))
                (+ (car L) (somme (cdr L))))
          (else (somme (cdr L))))))
```


Pourquoi une version en profondeur ?

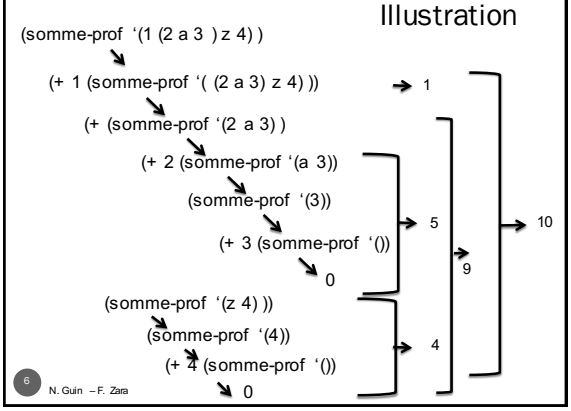
- (somme '(1 2 3 z 4)) → 10
- (somme '(1 (2 a 3) z 4)) → 5
- (somme '(1 (2 (3 b 6) 7) z 4) → 5

- La fonction somme effectue seulement la somme des nombres non imbriqués dans des listes. Nous aimerions une fonction somme-prof qui permette les appels suivants :
- (somme-prof '(1 2 3 z 4)) → 10
- (somme-prof '(1 (2 a 3) z 4)) → 10
- (somme-prof '(1 (2 (3 b 6) 7) z 4) → 23

Fonction somme : version en profondeur

```
(define somme-prof ; → nombre
  (lambda (L) ; L Liste
    (cond ((null? L) 0)
          ((number? (car L))
                (+ (car L) (somme-prof (cdr L))))
          ((list? (car L))
                (+ (somme-prof (car L))
                   (somme-prof (cdr L))))
          (else (somme-prof (cdr L))))))
```


Illustration



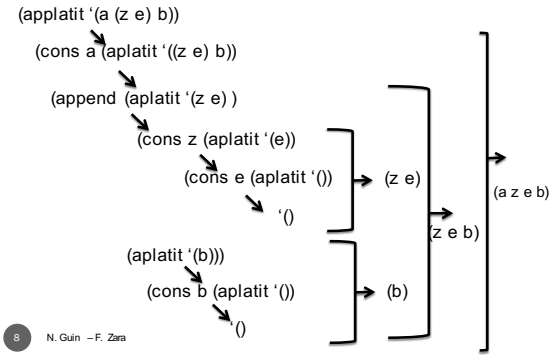
Deuxième exemple : la fonction « aplatit »

- Écrivons une fonction qui enlève toutes les parenthèses d'une liste quelconque

```
(define aplatit ; → liste d'atomes
   (lambda (L) ; L Liste
      (cond  ((null? L) '())
             ((list? (car L))
                    (append (aplatit (car L))
                            (aplatit (cdr L))))
             (else (cons (car L) (aplatit (cdr
 L)))))))
```

- (aplatit '(a (z e (a h) a b) i)) → (a z e a h a b i)

Illustration



Arbres

Arbres binaires
Représentation des arbres
Fonctions primitives sur les arbres
Parcours d'arbres
Arbres ordonnés

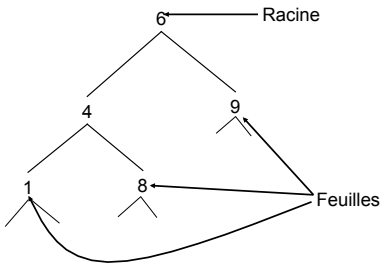
À quoi servent les arbres ?

- Les arbres, comme les listes, permettent de représenter un nombre variable de données
- Le principal avantage des arbres par rapport aux listes est qu'ils permettent de ranger les données de telle sorte que les recherches soient plus efficaces

Définition

- Un arbre est soit un nœud, soit un arbre vide
- Un nœud a des fils qui sont eux aussi des arbres
- Si tous les fils d'un nœud sont vides, alors le nœud est qualifié de feuille
- Les nœuds portent des valeurs, ce sont les données que l'on veut stocker
- Si tous les nœuds de l'arbre ont n fils, alors l'arbre est dit n-aire

Exemple d'arbre



Arbres binaires

- Un arbre binaire est :
  - soit l'arbre vide
  - soit un nœud qui a exactement deux fils (éventuellement vides)
- Pour manipuler les arbres binaires, on a besoin de primitives
  - d'accès,
  - de test
  - et de construction

Primitives sur les arbres binaires (1)

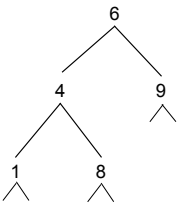
- Primitives d'accès
  - valeur : retourne la valeur d'un arbre non vide
  - fils-g : retourne le fils de gauche d'un arbre non vide
  - fils-d : retourne le fils de droite d'un arbre non vide
- Primitives de test
  - vide? : retourne vrai si un arbre donné est vide
  - arbre=? : retourne vrai si deux arbres donnés sont égaux

Primitives sur les arbres binaires (2)

- Primitives de construction
  - vide : retourne un arbre vide
  - cons-binaire : crée un arbre avec une valeur donnée et deux arbres donnés qui seront ses deux uniques fils

Exemples d'utilisation des primitives

Soit l'arbre a :



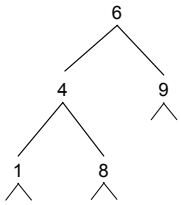
(valeur a) → 6
(valeur (fils-g a)) → 4
(valeur (fils-d (fils-g a))) → 8
(vide? a) → #f
(vide? (fils-d a)) → #f
(vide? (fils-g (fils-d a))) → #t

Représentation des arbres binaires

- Nous choisissons d'utiliser les listes pour représenter les arbres
- Un arbre vide sera représenté par la liste vide ()
- Un nœud sera une liste de 3 éléments
  - le car est sa valeur
  - le cadr son fils gauche
  - le caddr son fils droit

Exemple de représentation d'un arbre binaire

(define a '(6 (4 (1 () ()) (8 () ())) (9 () ())))



Définitions des primitives (1)

- valeur : retourne la valeur d'un arbre non vide
  (define valeur ; → atome
  (lambda (arbre) ; arbre non vide
  (car arbre)))
- fils-g : retourne le fils de gauche d'un arbre non vide
  (define fils-g ; → arbre
  (lambda (arbre) ; arbre non vide
  (cadr arbre)))

Définitions des primitives (2)

- fils-d : retourne le fils de droite d'un arbre non vide
  (define fils-d ; → arbre
  (lambda (arbre) ; arbre non vide
  (caddr arbre)))
- vide? : retourne vrai si un arbre donné est vide
  (define vide? ; → booléen
  (lambda (arbre) ; arbre
  (null? arbre)))

Définitions des primitives (3)

- arbre=? : retourne vrai si deux arbres donnés sont égaux
  (define arbre=? ; → booléen
    (lambda (arbre1 arbre2) ; arbres
      (equal? arbre1 arbre2)))
- vide : retourne un arbre vide
  (define vide ; → arbre
    (lambda ()
      '()))

Définitions des primitives (4)

- cons-binaire : crée un arbre avec une valeur donnée et deux arbres donnés qui seront ses deux uniques fils
  (define cons-binaire ; → arbre
   (lambda (val fg fd); val atome, fg et fd arbres
      (list val fg fd)))

Exemples

- (define a '(6 (4 (1 () ()) (8 () ())) (9 () ())))
- (define b (fils-g a))
- b
  → (4 (1 () ()) (8 () ()))
- (cons-binaire 2 b (fils-d a))
  → (2 (4 (1 () ()) (8 () ())) (9 () ()))

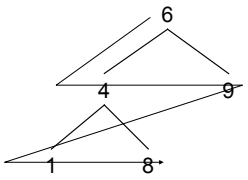
Pourquoi utiliser des primitives ?

- Pourquoi utiliser (vide) au lieu de () et fils-g au lieu de cadr ?
- Si on décide de changer la représentation des arbres :
  - sans primitives, il faut réécrire toutes les fonctions sur les arbres
  - avec primitives, il suffit de modifier les primitives

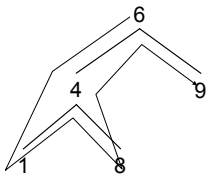
Parcours d'arbres

- Un arbre contient un ensemble de données
- Pour utiliser ces données, il faut parcourir l'arbre : en profondeur ou en largeur

Parcours en largeur



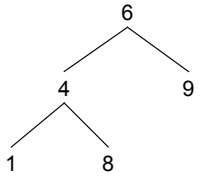
Parcours en profondeur : un exemple



Parcours en profondeur : principe

- Parcourir un arbre en profondeur consiste à passer ses nœuds en revue, en commençant toujours par le même fils, et en descendant le plus profondément possible dans l'arbre
- Lorsque l'on arrive sur un arbre vide, on remonte jusqu'au nœud supérieur et on redescend dans le fils encore inexploré

3 parcours en profondeur



- Parcours infixe : Fils-g Valeur Fils-d
  → 1 4 8 6 9
- Parcours préfixe : Valeur Fils-g Fils-d
  → 6 4 1 8 9
- Parcours postfixe : Fils-g Fils-d Valeur
  → 1 8 4 9 6

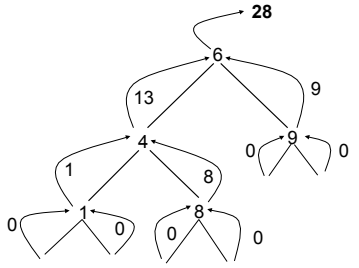
Pour écrire une fonction qui effectue un parcours en profondeur

- Pour écrire une fonction f, sur un arbre A
  - Si A est vide, on retourne une valeur constante, généralement l'élément neutre de f
  - Si A n'est pas vide :
    - on rappelle f sur les deux fils de A, ce qui retourne deux résultats : Rg et Rd
    - puis on retourne un résultat qui ne dépend que de Rg, Rd et de la valeur de la racine de A

Exemple : somme des valeurs d'un arbre

```
(define somme ; → nombre
  (lambda (A) ; A arbre de nombres
    (if (vide? A)
        0
        (+      (somme (fils-g A))
                (somme (fils-d A))
                (valeur A)))))
```

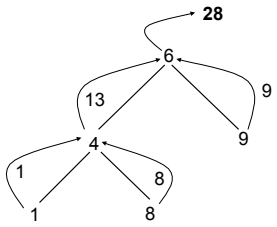

Fonctionnement sur un exemple



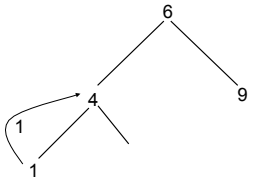
Modification du cas d'arrêt

```
(define somme ; → nombre
 (lambda (A) ; A arbre de nombres
  (if (feuille? A)
      (valeur A)
      (+      (somme (fils-g A))
              (somme (fils-d A))
              (valeur A)))))
```


Premier test



Deuxième test



Moralité

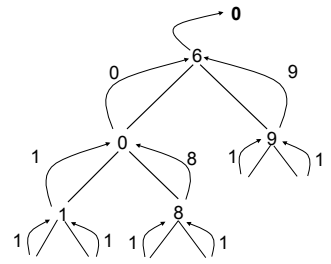
- Il faut toujours tester les fonctions sur un arbre dont un nœud n'a qu'un seul fils
- Il faut toujours prévoir le cas d'arrêt correspondant à l'arbre vide

Parcours partiels

- Il est parfois souhaitable d'arrêter le parcours même si tous les nœuds n'ont pas été passés en revue
- Exemple : produit des valeurs d'un arbre

```
(define produit ; → nombre
 (lambda (A) ; A arbre de nombres
  (if     (vide? A)
          1
  (* (produit (fils-g A))  (produit (fils-d A)) (valeur A)))))
```

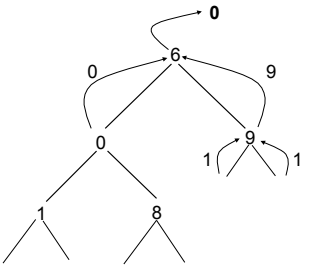

Premier test



Modification de la fonction

```
(define produit ; → nombre
 (lambda (A) ; A arbre de nombres
  (cond          ((vide? A) 1)
                 ((= 0 (valeur A)) 0)
                 (else
  (* (produit (fils-g A))  (produit (fils-d A)) (valeur A))))))
```


Deuxième test



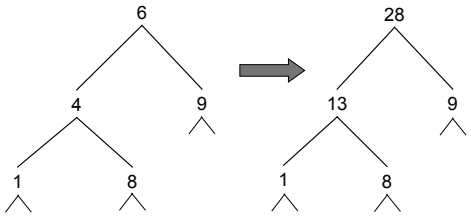
Modification et création d'arbres

- Exemple : écrire une fonction qui ajoute 1 à tous les nœuds d'un arbre qui contient des nombres
- Il ne s'agit pas d'une modification (ajouter 1), mais d'une création : écrire une fonction qui retourne un arbre identique à celui passé en argument, mais dans lequel on a ajouté 1 à tous les nœuds

Fonction ajoute1

```
(define ajoute1 ; → arbre
 (lambda (A) ; A arbre de nombres
  (if (vide? A)
     A
     (cons-binaire      (+ 1 (valeur A))
                        (ajoute1 (fils-g A))
                        (ajoute1 (fils-d A))))))
```


Somme des valeurs des fils



Première solution : utiliser la fonction somme

```
(define somme-fils ; → arbre
 (lambda (a) ; a arbre de nombres
  (if (vide? a)
     (vide)
     (cons-binaire
        (somme a)
        (somme-fils (fils-g a))
        (somme-fils (fils-d a))))))
```

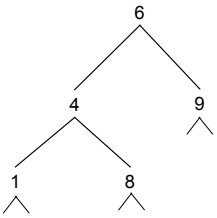
Réflexion sur cette fonction

- La complexité de cette fonction est beaucoup trop grande
- Il faut utiliser la valeur de la racine du résultat de l'appel récursif sur les fils : ils contiennent déjà la somme des valeurs de tous les nœuds de chacun des fils

Modification de la fonction

```
(define somme-fils ; → arbre
  (lambda (A) ; A arbre de nombres
    (if  (vide? A)
         (vide)
         (let   ((g (somme-fils (fils-g A)))
         (d (somme-fils (fils-d A))))
     (cons-binaire       (+ (valeur A) (valeur g)
                                       (valeur d))
                         g d)))))
```

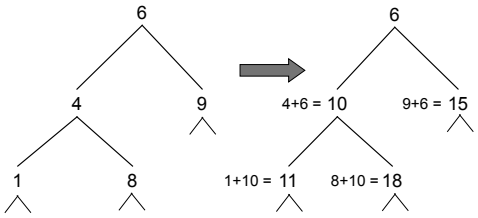

Test de la fonction



Correction de la fonction

```
(define somme-fils ; → arbre
  (lambda (A) ; A arbre
    (if    (vide? A))
           (vide)
           (let   ((g (somme-fils (fils-g A)))
           (d (somme-fils (fils-d A))))
     (cons-binaire  (+ (valeur A)
                       (if (vide? g) 0 (valeur g))
                       (if (vide? d) 0 (valeur d)))
                    g d)))))
```


Somme des valeurs des pères



Calcul en remontant ou en descendant

- Dans toutes les fonctions précédemment écrites, le résultat dépendait des fils
  ⇒ calcul en remontant
- Ici, le résultat dépend du père
  ⇒ calcul en descendant
  ⇒ paramètre supplémentaire pour passer le résultat du père au fils

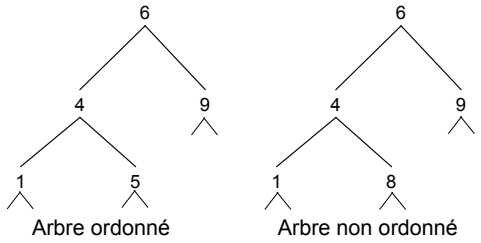
Fonction somme-pere

- (define somme-pere ; → arbre
  (lambda (A) ; A arbre de nombres
  (somme-pere2 A 0)))
- (define somme-pere2 ; → arbre
  (lambda (A n) ; A arbre de nb, n nombre
  (if (vide? A)
  (vide)
  (let ((v (+ (valeur A) n)))
  (cons-binaire v
  (somme-pere2 (fils-g A) v)
  (somme-pere2 (fils-d A) v))))))

Arbres de recherche (ou ordonnés)

- Les valeurs des nœuds peuvent être ordonnées
- En chaque nœud de l'arbre, la valeur du nœud est :
  - supérieure à toutes celles de son fils gauche
  - inférieure à toutes celles de son fils droit

Exemples



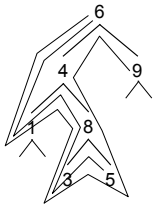
Recherche d'un élément dans un arbre binaire quelconque (1)

- On souhaite écrire une fonction qui teste l'appartenance d'une valeur V à un arbre A
- Principe : tant qu'on n'a pas trouvé la valeur V, il faut comparer V avec toutes les valeurs de l'arbre A

Recherche d'un élément dans un arbre binaire quelconque (2)

- Algorithme :
  - Cas d'arrêt :
    - Si A est vide Alors Retourne Faux
    - Si valeur(A)=V Alors Retourne Vrai
  - Appels récursifs :
    - Chercher V dans fils-gauche(A)
    - Puis si on n'a toujours pas trouvé V, chercher V dans fils-droit(A)

Exemple



Recherche fructueuse :
Chercher 5

Cas le pire
Recherche infructueuse :
Chercher 7

Complexité au pire :
nombre de nœuds de l'arbre

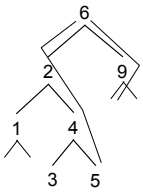
Recherche d'un élément dans un arbre binaire ordonné (1)

- Principe : utiliser le fait que l'arbre est ordonné pour choisir dans quelle branche de l'arbre chercher

Recherche d'un élément dans un arbre binaire ordonné (2)

- Algorithme :
  - Cas d'arrêt :
    - Si A est vide Alors Retourne Faux
    - Si valeur(A)=V Alors Retourne Vrai
  - Appels récursifs :
    - Si V>valeur(A) Alors chercher V dans fils-droit(A)
    - Si V<valeur(A) Alors chercher V dans fils-gauche(A)

Exemple



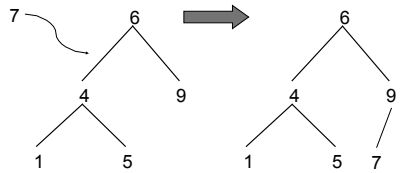
Recherche fructueuse : Chercher 5

Recherche infructueuse : Chercher 7

Complexité au pire : hauteur de l'arbre

Insertion dans un ABR

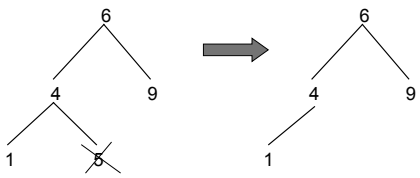
- Principe : on insère aux feuilles



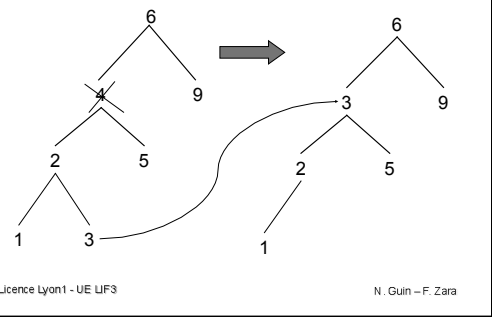
Suppression dans un ABR

- Pour supprimer la valeur V dans un ABR
  - Si V est une feuille, alors on supprime la feuille
  - Sinon on remplace la valeur V par la valeur V' qui lui est immédiatement inférieure (ou immédiatement supérieure), de manière à respecter l'ordre, puis on supprime V' qui est une feuille
  - V' est le plus grand élément du fils gauche de V (ou le plus petit élément de son fils droit)

Exemple



Exemple



Programmation d'ordre supérieur

Fonctions en argument
Fonctions en résultat
Abstraction
Map
Apply

Retour au B.A. BA

- Fonction Somme

```
(define somme ; → nb
  (lambda (L); liste nb
      (if (null? L)
        0
        (+ (car L) (somme
(cdr L))))))
```

- Fonction Produit

```
(define produit ; → nb
  (lambda (L); liste nb
      (if (null? L)
        1
        (* (car L) (produit
(cdr L))))))
```


Essayons de généraliser

- Fonction CalculSurListe

```
(define CalculSurListe ; → nombre
  (lambda (L ) ; L liste de nombres
      (if (null? L)
        ValeurSiVide ; élément neutre
        (FonctionDeCalcul (car L)
            (CalculSurListe(cdr L)..)))))
```


Passer une fonction en argument

- En Scheme, il est possible d'utiliser des arguments de type « fonction »
- Exemple : une fonction qui, étant donnés une fonction et un argument, applique deux fois la fonction à l'argument

Fonction applique2fois

- (define applique2fois ; → résultat de f
  (lambda (f n) ; f fonction, n arg^{t} de f
  (f (f n))))
- Exemples :
  (applique2fois sqrt 16) → 2
  (applique2fois cdr '(a b c d)) → (c d)
- Comment mémoriser la fonction « applique2fois sqrt » ?

Fonction en résultat (1)

- On voudrait écrire :

```
(define doubleRacine
    (DoublerFonction sqrt))
```

- Pour ensuite utiliser :
  (doubleRacine 16) → 2
- Il faut donc définir la fonction DoublerFonction, qui retourne une fonction

Fonction en résultat (2)

- Qu'est ce qu'une fonction ?
  - Une lambda-expression
- Une fonction qui retourne une fonction retourne donc une lambda-expression
- (define DoublerFonction ; → fonction
  (lambda (f) ; f fonction à répéter
  (lambda (n) ; résultat de
  (f (f n))))) ; DoublerFonction

Autre exemple : générer une fonction d'addition

- (define add-gen ; → fonction qui ajoute x à un nombre
  (lambda (x) ; x nombre
  (lambda (y)
  (+ x y))))
- (define add-5 (add-gen 5))
- (add-5 9) → 14
- ((add-gen 5) 3) → 8

Abstraction

- Reprenons notre fonction pour généraliser les calculs sur une liste de nombres
- (define CalculSurListe ; → nombre
  (lambda (L f n) ; L liste de nombres,
  f fonction de 2 nb, n nombre
  (if (null? L)
  n
  (f (car L) (CalculSurListe (cdr L) f n)))))

Utilisation de la fonction abstraite

- (define somme ; → nombre
  (lambda (L) ; L liste de nombres
  (CalculSurListe L + 0)))
- (define produit ; → nombre
  (lambda (L) ; L liste de nombres
  (CalculSurListe L * 1)))

Un autre exemple d'abstraction (1)

- Une fonction qui ajoute x à tous les éléments d'une liste de nombres
- (define ajoute-x ; → liste de nombres
  (lambda (x L) ; x nb, L liste de nb
  (if (null? L)
  '()
  (cons (+ x (car L))
  (ajoute-x x (cdr L))))))

Un autre exemple d'abstraction (2)

- Une fonction qui multiplie par 2 tous les éléments d'une liste de nombres
- (define double ; → liste de nombres
  (lambda (L) ; L liste de nb
  (if (null? L)
  '()
  (cons (* 2 (car L))
  (double (cdr L))))))

Un autre exemple d'abstraction (3)

- Généralisation : appliquer une fonction à tous les éléments d'une liste
- (define applique-à-tous ; → liste
  (lambda (f L) ; f fonction unaire, L liste
    (if (null? L)
      '()
      (cons (f (car L))
        (applique-à-tous f (cdr L))))))

Un autre exemple d'abstraction (4)

- (define ajoute-x ; → liste de nombres
  (lambda (x L) ; x nb, L liste de nb
    (applique-à-tous
      (lambda (y) (+ y x)) L)))
- (define double ; → liste de nombres
  (lambda (L) ; L liste de nb
    (applique-à-tous
      (lambda (x) (* x 2)) L)))

Abstraction des parcours d'arbres (1)

- (define somme ; → nombre
  (lambda (A) ; A arbre de nb
    (if (vide? A)
      0
      (+ (valeur A)
        (somme (fils-g A))
        (somme (fils-d A))))))

Abstraction des parcours d'arbres (2)

- (define produit ; → nombre
  (lambda (A) ; A arbre de nb
    (if (vide? A)
      1
      (* (valeur A)
        (produit (fils-g A))
        (produit (fils-d A))))))

Abstraction des parcours d'arbres (3)

- (define absArbres ; → nombre
  (lambda (f n A) ; A arbre de nb,
    n nombre, f fonction ternaire
    (if (vide? A)
      n
      (f (valeur A)
        (absArbres f n (fils-g A))
        (absArbres f n (fils-d A))))))

Abstraction des parcours d'arbres (4)

- (define somme ; → nombre
  (lambda (A) ; A arbre de nb
    (absArbre + 0 A)))
- (define produit ; → nombre
  (lambda (A) ; A arbre de nb
    (absArbre * 1 A)))

Abstraction des parcours d'arbres (5)

- (define maximum ; → nombre
  (lambda (A) ; A arbre de nb positifs
  (absArbre max 0 A)))
- (define compter ; → nombre
  (lambda (A) ; A arbre
  (absArbre
  (lambda (x y z) (+ 1 y z))
  0 A)))

La fonction Map

- La fonction applique-à-tous est tellement utile qu'elle est prédéfinie en Scheme
- Elle porte le nom map et est en fait une version plus générale de applique-à-tous

Exemples (1)

- Map s'utilise comme applique-à-tous
- (map (lambda (x) (+ x 10)) '(10 3 100 9 64))
  → (20 13 110 19 74)
- (map number? '(10 z e (1 2) 3 g 4))
  → (#t #f #f #f #t #f #t)
- (map (lambda (z) (cons z '(k))) '(1 2 a b))
  → ((1 k) (2 k) (a k) (b k))

Exemples (2)

- Map est plus générale qu'applique-à-tous car la fonction à appliquer peut prendre plus d'un argument
- Il faut alors donner à cette fonction autant de listes qu'elle doit avoir d'arguments, toutes les listes devant être de même longueur
- (map (lambda (a b) (+ a b)) '(1 2 3) '(10 20 30)) → (11 22 33)
- (map + '(1 2 3) '(10 20 30)) → (11 22 33)
- (map list '(1 2) '(a z) '(q s)) → ((1 a q) (2 z s))

La fonction Apply

- (apply f l) applique la fonction f à l'ensemble des éléments de la liste l
- (apply + '(1 2 3 4)) → 10
- (apply (lambda (z) (cons z '(k))) '((1 2 a b)))
  → ((1 2 a b) k)
- La fonction apply n'est pas très utile en elle-même
- Elle sert pour exploiter le résultat d'un map, qui est toujours une liste

Map-Apply : un exemple

- Une fonction pour calculer le produit scalaire de deux vecteurs :
  $L_1 = '(x_1\ x_2 \ldots x_n)$
  $L_2 = '(y_1\ y_2 \ldots y_n)$
  produit scalaire = $x_1y_1 + x_2y_2 + \ldots + x_ny_n$
- (define scalaire ; → nombre
  (lambda (L1 L2) ; listes de nb
  (apply + (map * L1 L2))))

Définitions « formelles »

- Soit f_1 une fonction unaire
  (map f_1 '(x_1 … x_n)) → ($f_1(x_1)$ … $f_1(x_n)$)
- Soit f_2 une fonction binaire
  (map f_2 '(x_1 … x_n) '(y_1 … y_n))
  → ($f_2(x_1,y_1)$ … $f_2(x_n,y_n)$)
- Soit f_3 une fonction ternaire
  (map f_3 '(x_1 … x_n) '(y_1 … y_n) '(z_1 … z_n))
  → ($f_3(x_1,y_1,z_1)$ … $f_3(x_n,y_n,z_n)$)
- … et ainsi de suite
- Soit f_n une fonction n-aire
  (apply f_n '(x_1 … x_n)) → $f_n(x_1, \ldots, x_n)$

Map-Apply : les différences (1)

MAP

- Prend autant de listes que l'arité de la fonction, toutes les listes étant de même longueur

APPLY

- Prend une liste comme argument, la longueur de cette liste étant égale à l'arité de la fonction

Map-Apply : les différences (2)

MAP

- Applique la fonction à chaque élément de la liste (ou des listes)
- Retourne toujours une liste de résultats du type de celui de la fonction

APPLY

- Applique la fonction à l'ensemble des éléments de la liste
- Retourne un résultat du type de celui de la fonction

Listes d'associations

Une autre structure de données

Définitions

- Une liste d'associations est une liste de couples (les associations)
- Chaque association est un couple (ou paire) formé par une clef et une valeur

$$`((clef_1 . valeur_1)(clef_2 . valeur_2)\ldots(clef_n . valeur_n))$$

- Cette structure de données permet de chercher une valeur à partir d'une clef

Primitives

- Rechercher dans la liste une association à partir d'une clef
- Rechercher dans la liste la valeur d'une association à partir d'une clef
- Ajouter une association à une liste d'association
- Enlever d'une liste les associations qui ont une clef donnée
- Remplacer dans la liste la valeur correspondant à une clef

Exemples d'utilisation (1)

(define vitesses '((tortue . 15) (levrier . 60)
(guepard . 115) (lapin . 60) (autruche . 90)))

(donne-paire vitesses 'lapin 'pas-trouve)
→ (lapin . 60)
(donne-paire vitesses 'souris 'pas-trouve)
→ pas-trouve
(donne-valeur vitesses 'autruche 'pas-trouve)
→ 90

Exemples d'utilisation (2)

(ajoute-paire vitesses 'souris 25)
→ ((souris . 25) (tortue . 15) (levrier . 60) (guepard . 115) (lapin . 60) (autruche . 90))

(remplace-valeur vitesses 'guepard 130)
→((tortue . 15) (levrier . 60) (guepard . 130) (lapin . 60) (autruche . 90))

(enleve-clef vitesses 'guepard)
→ ((tortue . 15) (levrier . 60) (lapin . 60) (autruche . 90))

Définition des primitives (1)

```
(define donne-paire ; → couple ou valeur par defaut
 (lambda (L  Clef  ValDef) ; alist * type clef * type quelconque
  (cond ((null? L) ValDef)
        ((eq? (caar L) Clef) (car L))
        (else (donne-paire (cdr L) Clef  ValDef)))))
```

N.B. Une fonction prédéfinie en scheme a presque le même comportement :

(assoc 'levrier vitesses) →(levrier . 60)

La valeur par défaut de la fonction assoc est #f

Définition des primitives (2)

```
(define donne-valeur ; → type valeur
  (lambda (L  Clef  ValDef)  ; alist * type clef * type quelconque
    (let ((res (donne-paire  L  Clef  ValDef))) ; res : paire
      (if (eq? res ValDef)
          ValDef
          (cdr res))))) ; cdr res : second élement de res
```

```
(define ajoute-paire ; → alist
  (lambda (L  Clef  Valeur) ; alist * type clef * type valeur
    (cons (cons Clef  Valeur) L)))
```


Définition des primitives (3)

```
(define enleve-clef ; → alist
  (lambda (L  Clef) ; alist * type clef
    (cond ((null? L) '())
          ((eq? (caar L) Clef) (enleve-clef (cdr L) Clef))
          (else (cons (car L) (enleve-clef (cdr L) Clef))))))
```

N.B. Cette fonction enlève de la liste d'association tous les couples de la Clef donnée

Définition des primitives (4)

```
(define remplace-valeur ; → alist
  (lambda (L  Clef  Valeur) ; alist * type clef * type valeur
    (cond ((null? L) (ajoute-paire '() Clef  Valeur))
          ((eq? Clef (caar L)) (ajoute-paire (cdr L) Clef  Valeur))
          (else (cons (car L) (remplace-valeur (cdr L) Clef  Valeur))))))
```


Un autre exemple

```
(define etudiants '((10302450 . (Jean Sauvert L2))
        (10405672 . (Sylvie Perrin L1)) (10304455 . (Aziz Bourras L1))))

(donne-valeur etudiants 10403302 'pas-trouve)
   → pas-trouve
(donne-valeur etudiants 10405672 'pas-trouve)
   → (sylvie perrin l1)
(remplace-valeur etudiants 10405672 '(sylvie perrin L2))
   → ((10302450 . (jean sauvert l2)) (10405672 . (sylvie perrin l2))
  (10304455 . (aziz bourras l1)))
```

Logique

Logique des propositions
Algèbre de Boole
Méthodes de simplification des fonctions booléennes

Objectifs

- Traiter formellement les notions de vérité et de fausseté
- Formaliser ce qu'on appelle le « raisonnement logique » ou la « déduction logique »

Exemple

- Tous les mardis je vais au cinéma
- Ce soir je vais au cinéma
- Quel jour sommes-nous ?

Une énigme policière

- Un meurtre a été commis au laboratoire, le corps se trouve dans la salle de conférences…
- On dispose des informations suivantes :
  - La secrétaire déclare qu'elle a vu l'ingénieur dans le couloir qui donne sur la salle de conférences
  - Le coup de feu a été tiré dans la salle de conférences, on l'a donc entendu de toutes les pièces voisines
  - L'ingénieur affirme n'avoir rien entendu
- On souhaite démontrer que si la secrétaire dit vrai, alors l'ingénieur ment

Logique des propositions : syntaxe

- On définit :
  - Les propositions : a, b, c, …
  - Les constantes : Vrai et Faux
  - Les connecteurs :
    - $\wedge$ (conjonction)
    - $\vee$ (disjonction)
    - $\neg$ (négation)
    - $\supset$ (implication)

Construction d'une formule

- Une proposition est une formule
- Si X et Y sont des formules, alors $\neg X$, $X \vee Y$, $X \wedge Y$, $X \supset Y$ sont des formules
- On utilise de plus les parenthèses pour lever des ambiguïtés
- Exemples:
  - $a \wedge (c \vee \neg d)$
  - $(a \vee b) \supset \neg c$

Logique des propositions : sémantique

- Les formules sont interprétées dans {Vrai,Faux}
- On définit l'interprétation associée à chaque connecteur grâce aux tables de vérité

L'opérateur ET

X	Y	X ∧ Y
Vrai	Vrai	Vrai
Vrai	Faux	Faux
Faux	Vrai	Faux
Faux	Faux	Faux

Les deux doivent être vrais pour que le ET soit vrai

L'opérateur OU

X	Y	X ∨ Y
Vrai	Vrai	Vrai
Vrai	Faux	Vrai
Faux	Vrai	Vrai
Faux	Faux	Faux

Il suffit que l'un des deux soit vrai pour que le OU soit vrai

L'opérateur NON

X	¬X
Vrai	Faux
Faux	Vrai

L'opérateur implique

X	Y	X ⊃ Y
Vrai	Vrai	Vrai
Vrai	Faux	Faux
Faux	Vrai	Vrai
Faux	Faux	Vrai

X ⊃Y signifie que si X est vrai alors Y est vrai
Le faux implique n'importe quoi

Autre définition de l'implication

- On peut aussi définir l'implication en disant : X ⊃Y = ¬X∨Y
- On retrouve la même table de vérité :

X	Y	¬X	¬X∨Y
Vrai	Vrai	Faux	Vrai
Vrai	Faux	Faux	Faux
Faux	Vrai	Vrai	Vrai
Faux	Faux	Vrai	Vrai

Définition de l'équivalence

- $X \Leftrightarrow Y = (X \supset Y) \wedge (Y \supset X)$
- X est une condition nécessaire et suffisante pour Y, et on dit X si et seulement si Y
- Y est une condition nécessaire et suffisante pour X, et on dit Y si et seulement si X

Conditions nécessaires et suffisantes

- Lorsqu'on écrit $X \supset Y$:
  - X est une condition suffisante de Y, et on dit X seulement si Y
  - Y est une condition nécessaire de X, et on dit Y si X
- Dérivable ⊃ continue
- Mardi ⊃ cinéma

Définition du OU-exclusif

- Le OU logique est dit *inclusif*. Le OU exclusif est celui du langage courant : fromage *ou* dessert.

X	Y	X OU-ex Y
V	V	F
V	F	V
F	V	V
F	F	F

Le OU-ex est vrai si l'un des deux est vrai mais pas les deux en même temps
(c'est le contraire de l'équivalence)

Propriétés des formules

- Une formule est valide si elle est toujours vraie (quelque soit l'interprétation)
- Une formule est consistante s'il existe une interprétation dans laquelle elle est vraie. Elle est inconsistante dans le cas contraire
- Problème : étant donnée une formule, est-elle valide ? consistante ?
- Exemple : que dire de la formule
  $(a \supset b) \supset (\neg b \supset \neg a)$

Dressons la table de vérité

a	b	a⊃b	¬b	¬a	¬b⊃¬a	(a⊃b)⊃(¬b⊃¬a)
V	V	V	F	F	V	V
V	F	F	V	F	F	V
F	V	V	F	V	V	V
F	F	V	V	V	V	V

Conclusion

- Quelles que soient les valeurs de a et b, cette formule est toujours vraie. Elle est donc valide.
- On peut montrer également que $(\neg a \supset \neg b) \supset (b \supset a)$
- C'est la contraposée des mathématiques

Règles de transformation (1)

- Toutes les formules qui suivent sont valides. Elles sont utiles pour simplifier des formules. Elles peuvent se démontrer en établissant leurs tables de vérité.
- X∨¬X = V (tiers exclu)
- X∧¬X = F (contradiction)
- ¬¬X = X (involution)
- X∨X = X∧X = X (idempotence)

Règles de transformation (2)

- ¬V = F, ¬F = V
- F∧X = F, V∧X = X, F∨X = X, V∨X = V
  - Faux est élément neutre pour le OU et absorbant pour le ET
  - Vrai est élément neutre pour le ET et absorbant pour le OU
- X∧(X∨Y) = X, X∨(X∧Y) = X (absorption)
- X∨(¬X∧Y) = X∨Y
- X⊃Y = ¬X ∨ Y

Règles de transformation (3)

- Lois de De Morgan :
  - ¬(X∨Y) = ¬X ∧ ¬Y
  - ¬(X∧Y) = ¬X ∨ ¬Y
- ((X⊃Y) ∧X) ⊃Y (modus ponens)
- ((X⊃Y) ∧¬Y) ⊃ ¬X (modus tollens)
- (X⊃Y) = (¬Y ⊃ ¬X) (contraposition)

Règles de transformation (4)

- Commutativité et associativité de ∨ et ∧
  - X∨Y=Y∨X, X∧Y=Y∧X
  - X∨(Y∨Z)=(X∨Y)∨Z=X∨Y∨Z
  - X∧(Y∧Z)=(X∧Y)∧Z=X∧Y∧Z
- Distributivité de ∨ par rapport à ∧ et de ∧ par rapport à ∨
  - X∨(Y∧Z)=(X∨Y)∧(X∨Z), X∧(Y∨Z)=(X∧Y)∨(X∧Z)
- Transitivité de ⊃
  - ((X⊃Y)∧(Y⊃Z)) ⊃ (X⊃Z)

Règles de transformation (5)

- F⊃X = V
- V⊃X = X
- X⊃F = ¬X
- X⊃V = V

Exemple d'application des règles de transformation

- ¬(p⊃q)
  = ¬(¬p∨q)
  = ¬¬p∧¬q
  = p∧¬q

Retour sur l'énigme policière

- Un meurtre a été commis au laboratoire, le corps se trouve dans la salle de conférences…
- On dispose des informations suivantes :
  - La secrétaire déclare qu'elle a vu l'ingénieur dans le couloir qui donne sur la salle de conférences
  - Le coup de feu a été tiré dans la salle de conférences, on l'a donc entendu de toutes les pièces voisines
  - L'ingénieur affirme n'avoir rien entendu
- On souhaite démontrer que si la secrétaire dit vrai, alors l'ingénieur ment

Formalisation en calcul des propositions

- p : la secrétaire dit vrai
- q : l'ingénieur était dans le couloir au moment du crime
- r : l'ingénieur était dans une pièce voisine de la salle de conférences
- s : l'ingénieur a entendu le coup de feu
- t : l'ingénieur dit vrai

Résolution de l'énigme

- Les informations de l'énoncé se traduisent par les implications :
  $p \supset q$, $q \supset r$, $r \supset s$, $t \supset \neg s$
- Il s'agit de prouver la validité de la formule :
  $(p \supset q \wedge q \supset r \wedge r \supset s \wedge t \supset \neg s) \supset (p \supset \neg t)$

Démonstration

$(p \supset q \wedge q \supset r \wedge r \supset s \wedge t \supset \neg s) \supset (p \supset \neg t)$

- La formule ne peut être fausse que si
  - $(p \supset \neg t)$ est faux, soit p et t vrais
  - la prémisse est vraie, soit toutes les implications vraies
- Comme t doit être vrai, s doit être faux, donc r faux, donc q faux, donc p faux, et il y a contradiction

Applications de la logique en informatique

- En algorithmique : nier des conditions, spécifier des invariants, des pré-conditions et des post-conditions
- En architecture : réalisation de toute fonction de traitement de code binaire à partir des seules portes logiques binaires OU, ET, NON (électronique numérique)
- En base de données : interprétation logique des BD, logique pour l'interrogation des BD, expression des contraintes d'intégrité, BD déductives
- En Intelligence Artificielle : représentation des connaissances, systèmes experts

Algèbre de Boole

- En informatique, on utilise plutôt 1 (tension haute) à la place de Vrai, et 0 (tension basse) à la place de Faux

A	B	$\neg A$	$A \wedge B$	$A \vee B$
1	1	0	1	1
1	0	0	0	1
0	1	1	0	1
0	0	1	0	0

Les connecteurs en Algèbre de Boole

- On obtient les relations suivantes :
  - ¬X=1-X qu'on notera $\overline{X}$
  - X∧Y=X.Y
  - X∨Y=min(X+Y,1) qu'on notera X+Y
- On remplace X⊃Y par ¬X∨Y, soit $\overline{X}$+Y

Ecriture des règles de transformation en algèbre de Boole (1)

- X+$\overline{X}$=1, X.$\overline{X}$=0 (tiers exclu, contradiction)
- $\overline{\overline{X}}$=X ,X+X=X, X.X=X (involution, idempotence)
- $\overline{0}$=1, $\overline{1}$=0
- X+0=X, X.0=0, X+1=1, X.1=X (éléments neutres et absorbants)
- X.(X+Y)=X, X+X.Y=X (absorption)
- X+$\overline{X}$.Y=X+Y
- $\overline{X+Y}$=$\overline{X}$.$\overline{Y}$, $\overline{X.Y}$=$\overline{X}$+$\overline{Y}$ (De Morgan)

Ecriture des règles de transformation en algèbre de Boole (2)

- X+Y=Y+X, X.Y=Y.X (commutativité)
- X+(Y+Z)=(X+Y)+Z=X+Y+Z(associativité)
- X.(Y.Z)=(X.Y).Z=X.Y.Z (associativité)
- X.(Y+Z)=X.Y+X.Z (distributivité)
- X+Y.Z=(X+Y).(X+Z) (distributivité)

Remarque : les règles souvent duales, en permutant + et . d'une part et 1 et 0 d'autre part

Fonctions booléennes

- En algèbre de boole, on parle de fonctions booléennes plutôt que de formules.
- Exemple : F = a(b+$\overline{c}$)+bc$\overline{a}$
- On peut aussi définir une fonction booléenne à partir de sa table de vérité

Exemple : une fonction "majorité"

a	b	c	M
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1

M = abc+abc+abc+abc

Pourquoi simplifier une fonction booléenne ?

- Pour dresser plus facilement sa table de vérité afin de :
  - Déterminer la validité de la fonction
  - La comparer avec une autre fonction
- Pour concevoir un circuit intégré réalisant la fonction avec le moins de portes logiques possible

Exemple à l'aide des règles de l'algèbre de Boole

- Reprenons la fonction "majorité"
- M = abc+abc+abc+abc
  = bc(a+a)+a(bc+bc)
  = bc+a(bc+bc)
- Peut-on trouver plus simple ?

Deuxième exemple

- F = abc+a(bc+bc)
  = abc+ abc+abc
  = ab(c+c)+abc
  = ab+abc
  = a(b+bc)
  = a(b+c)

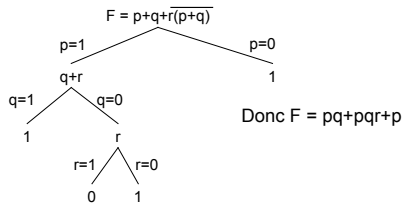
Méthodes de simplification des fonctions booléennes

- Deux méthodes permettent de simplifier plus efficacement des fonctions compliquées que la seule application des règles de l'algèbre de Boole.
- Les diagrammes de Quine
- Les tables de Karnaugh

Méthode des diagrammes de Quine

- Principe :
  - On choisit une des variables qui interviennent le plus souvent dans la fonction booléenne à simplifier
  - On considère le cas où elle vaut 0 et le cas où elle vaut 1
  - On simplifie les deux expressions obtenues
  - On itère le processus sur les deux expressions simplifiées

Exemple



Donc F = pq+pqr+p

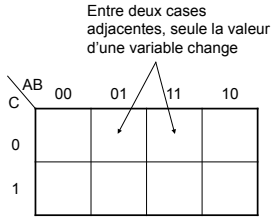
Utilisation des diagrammes de Quine

- Les diagrammes de Quine permettent de mettre une fonction booléenne sous la forme d'une somme de produits.
- Ils permettent aussi de vérifier la validité d'une expression booléenne :
  toutes les feuilles sont-elles égales à 1 ?

Méthode des tables de Karnaugh

- La fonction doit être en premier lieu exprimée comme une somme de produits. Pour ce faire, on utilise les règles de l'algèbre de Boole ou les diagrammes de Quine.
- On dresse ensuite une table de Karnaugh, qui est une table de vérité à deux dimensions. Sur chaque dimension on peut représenter les valeurs possibles de deux variables.

Table de Karnaugh à 3 variables



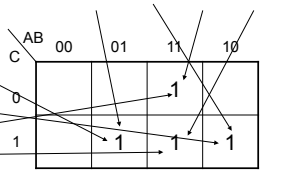
Remplir la table de Karnaugh

- On met 1 dans une case de la table si la fonction est vraie pour les valeurs des variables correspondant à cette case.
- On procède à des regroupements de 1 adjacents.
- On cherche à effectuer les groupements les plus grands afin de simplifier au maximum.
- Les regroupements sont des rectangles de 2^n termes.

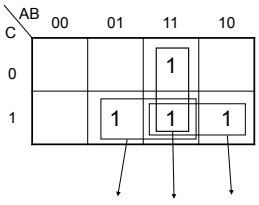
Retour sur la fonction « majorité »
1. Remplir la table

a	b	c	M
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1

M = abc+abc+abc+abc



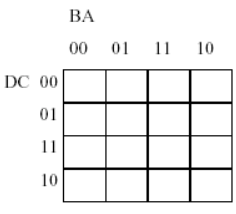
Retour sur la fonction « majorité »
2. Simplifier



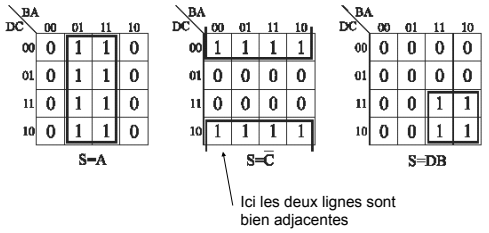
Donc F= BC + AB + AC

C'est plus simple que ce qu'on avait trouvé avec les règles de l'algèbre de Boole

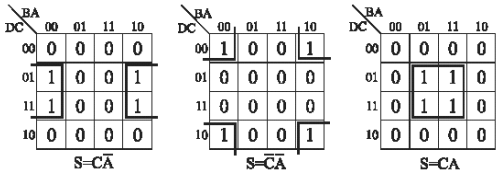
Table de Karnaugh à 4 variables



Exemples de simplification sur des tables de Karnaugh à 4 variables (1)



Exemples de simplification sur des tables de Karnaugh à 4 variables (2)



Et pour 5 variables ?