11X001 – Introduction à la programmation des algorithmes

3.4 Listes, Piles et Arbres

François Fleuret

https://fleuret.org/francois

27 Octobre, 2020

UNIVERSITÉ
DE GENÈVE

FACULTY OF SCIENCE

Structures récursifs

Il arrive fréquemment que l'on veuille définir une structure de données récursive.

Par exemple, on peut considérer qu'une liste d'entiers est:

- soit vide,
- soit composée d'un entier et d'une autre liste.

La liste $\{12, 99, 37\}$ peut ainsi être vue comme composée de l'entier 12 et de la liste $\{99, 37\}$, elle même composée de 99 et de la liste $\{37\}$, elle même composée de 37 et $\{\}$.

Structures récursives (cont.)

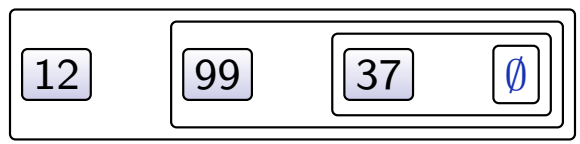
Il est donc naturel de vouloir définir une structure ayant des propriétés du même type qu'elle, comme une liste qui peut être composée d'une valeur et d'une liste.

En Swift, le choix naturel pour un type de la forme "X ou Y" est une énumération, qui peut être récursive.

Pour des raisons de représentation en mémoire, il faut indiquer avec le mot-clé `indirect` que l'une des valeurs d'une `enum` est paramétrée par le type lui-même.

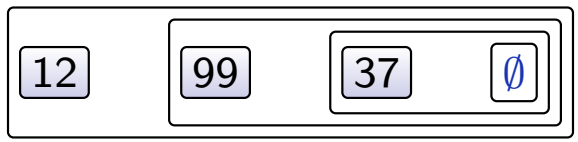
Exemple de liste chaînée (cont.)

La liste $\{12, 99, 37\}$ peut ainsi être représentée par:



Exemple de liste chaînée (cont.)

La liste $\{12, 99, 37\}$ peut ainsi être représentée par:



Mais en réalité la représentation de ce type d'objets en mémoire se fait toujours avec des références, d'où la nécessité d'indiquer `indirect`.

Exemple de liste chaînée

Exemple de liste chaînée

Nous pouvons définir une liste d'entiers récursivement avec:

```
indirect enum List {
  case empty
  case nonEmpty(Int, List)
}
```

Exemple de liste chaînée

Nous pouvons définir une liste d'entiers récursivement avec:

```
indirect enum List {
  case empty
  case nonEmpty(Int, List)
}

let list = List.nonEmpty(12, List.nonEmpty(99, List.nonEmpty(37, List.empty)))

print(list)
```

affiche

```
nonEmpty(12, lists.List.nonEmpty(99, lists.List.nonEmpty(37, lists.List.empty)))
```

Exemple de liste chaînée (cont.)

Calculer la somme des termes se met naturellement sous une forme récursive.

```
indirect enum List {
  case empty
  case nonEmpty(Int, List)
}
```

En effet, on a

- `sum(empty) = 0`, et
- `sum(nonEmpty(x, l)) = x + sum(l)`.

Exemple de liste chaînée (cont.)

Cette fonction se programme facilement à l'aide de `switch`:

```
indirect enum List {
  case empty
  case nonEmpty(Int, List)
}

func sum( _ list: List) -> Int {
  switch list {
  case List.empty:
    return 0
  case List.nonEmpty(let value, let tail):
    return value + sum(tail)
  }
}

let list = List.nonEmpty(12, List.nonEmpty(99, List.nonEmpty(37, List.empty)))

print(sum(list))
```

affiche 148.

Exemple de liste chaînée (cont.)

Ou récursive terminale

```
func sum( _ list: List) -> Int {
  func sumRec( _ list: List, _ s: Int) -> Int {
    switch list {
    case List.empty:
      return s
    case List.nonEmpty(let value, let tail):
      return sumRec(tail, s + value)
    }
  }
  return sumRec(list, 0)
}
```

Exemple de liste chaînée (cont.)

On peut programmer une fonction qui affiche une liste de manière plus lisible.

```
indirect enum List {
  case empty
  case nonEmpty(Int, List)
}
```

On voudrait que

- `printListRec(empty)` affiche `""`, et
- `printListRec(nonEmpty(x, l))` affiche `x`, un espace puis `l`.

Exemple de liste chaînée (cont.)

```
func printList( _ list: List) {
  func printListRec( _ list: List) {
    switch list {
    case List.empty:
      break
    case List.nonEmpty(let value, let tail):
      print(value, terminator: " ")
      printListRec(tail)
    }
  }

  print("{", terminator: " ")
  printListRec(list)
  print("}")
}

let l = List.nonEmpty(1, List.nonEmpty(2, List.nonEmpty(3, List.empty)))

printList(l)
```

affiche

```
{ 1 2 3 }
```

Exemple de liste chaînée (cont.)

Et on peut créer une liste à partir d'un tableau d'entiers.

```
func list( _ xs: [Int] ) -> List {
  func listRec( _ i: Int ) -> List {
    if i < xs.count {
      return List.nonEmpty(xs[i], listRec(i + 1))
    } else {
      return List.empty
    }
  }
  return listRec(0)
}

let l1 = list([ 1, 2, 3 ])
let l2 = list([ 4, 5 ])

printList(l1)
printList(l2)
```

affiche

```
{ 1 2 3 }
{ 4 5 }
```

Exemple de liste chaînée (cont.)

Pour faire une version récursive terminale, on doit partir de la fin.

```
func list( _ xs: [Int] ) -> List {
  func listRec( _ i: Int, _ l: List ) -> List {
    if i >= 0 {
      return listRec(i - 1, List.nonEmpty(xs[i], l))
    } else {
      return l
    }
  }
  return listRec(xs.count - 1, List.empty)
}
```

Exemple de liste chaînée (cont.)

Le même problème se pose si on utilise `reduce`:

```
let a = [ 2, 3, 4 ]
let l = a.reduce( List.empty, { l, v in List.nonEmpty(v, l) })
printList(l)
```

affiche

```
{ 4 3 2 }
```

Exemple de liste chaînée (cont.)

Le même problème se pose si on utilise `reduce`:

```
let a = [ 2, 3, 4 ]
let l = a.reduce( List.empty, { l, v in List.nonEmpty(v, l) })
printList(l)
```

affiche

```
{ 4 3 2 }
```

Il faut donc inverser le sens du parcours du tableau, ce qui est possible avec `reversed`:

```
let l = a.reversed().reduce( List.empty, { l, v in List.nonEmpty(v, l) })
```

Exemple de liste chaînée (cont.)

La concaténation peut aussi se faire fonctionnellement:

```
func concat( _ l1: List, _ l2: List) -> List {
  switch l1 {
  case List.empty:
    return l2
  case List.nonEmpty(let value, let tail):
    return List.nonEmpty(value, concat(tail, l2))
  }
}

let l1 = list([ 1, 2, 3 ])
let l2 = list([ 4, 5 ])

printList(l1)
printList(l2)
printList(concat(l1, l2))
```

affiche

```
{ 1 2 3 }
{ 4 5 }
{ 1 2 3 4 5 }
```

Exemple de liste chaînée (cont.)

Grace aux types génériques, nous pouvons généraliser nos listes à des types quelconques

```
indirect enum List<T> {
  case empty
  case nonEmpty(T, List)
}
```

Exemple de liste chaînée (cont.)

Avec par exemple

```
func list<T>( _ xs: [T] ) -> List<T> {
  func listRec( _ i: Int, _ l: List<T> ) -> List<T> {
    if i >= 0 {
      return listRec(i - 1, List.nonEmpty(xs[i], l))
    } else {
      return l
    }
  }
  return listRec(xs.count - 1, List.empty)
}

func concat<T>( _ l1: List<T>, _ l2: List<T>) -> List<T> {
  switch l1 {
  case List.empty:
    return l2
  case List.nonEmpty(let value, let tail):
    return List.nonEmpty(value, concat(tail, l2))
  }
}
```

Exemple de liste chaînée (cont.)

Nous pouvons écrire notre propre `map`!

```
func map<T, U>( _ f: (T) -> U, _ l: List<T> ) -> List<U> {
  switch l {
  case List.empty:
    return List<U>.empty
  case List.nonEmpty(let value, let tail):
    return List.nonEmpty(f(value), map(f, tail))
  }
}

let l = list([ "Paris", "Londres", "Berlin" ])
printList(l)
let m = map({ s in s.count }, l)
printList(m)
```

affiche

```
{ Paris Londres Berlin }
{ 5 7 6 }
```

Exemple de liste chaînée (cont.)

Et notre propre `filter`!

```
func filter<T>( _ f: (T) -> Bool, _ l: List<T> ) -> List<T> {
  switch l {
  case List.empty:
    return List<T>.empty
  case List.nonEmpty(let value, let tail):
    if f(value) {
      return List.nonEmpty(value, filter(f, tail))
    } else {
      return filter(f, tail)
    }
  }
}

let l = list([ "Paris", "Londres", "Berlin" ])
printList(l)
let m = filter({ s in s.count < 7 }, l)
printList(m)
```

affiche

```
{ Paris Londres Berlin }
{ Paris Berlin }
```

Exemple de liste chaînée (cont.)

Et notre propre `reduce`!

```
func reduce<T, U>( _ y: U, _ f: (U, T) -> U, _ l: List<T>) -> U {
  switch l {
  case List.empty:
    return y
  case List.nonEmpty(let value, let tail):
    return reduce(f(y, value), f, tail)
  }
}

let l = list([ "Paris", "Londres", "Berlin" ])

print(reduce(0, { (y, s) in y + s.count }, l))
```

affiche 18.

Exemple de liste chaînée (cont.)

Le principal défaut des listes chaînées est de ne pas permettre d'accéder directement au k-ième élément.

```
func get<T>( _ l: List<T>, _ k: UInt ) -> T? {
  switch l {
  case List.empty:
    return nil
  case List.nonEmpty(let value, let tail):
    if k == 0 {
      return value
    } else {
      return get(tail, k-1)
    }
  }
}

let u = list([ 100, 111, 122, 133, 144, 155, 166, 177, 188, 199, ])

print(get(u, 5) ?? 0)
```

affiche 155.

Extensions des listes

Pour être utilisable en pratique, un type liste est augmenté pour permettre d'insérer / supprimer des éléments et faire des concaténations rapidement.

Cela se fait généralement en ayant des références vers les éléments suivants et précédents.

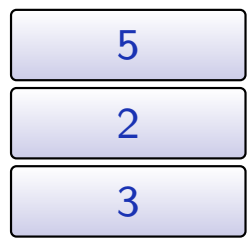
En pratique la plupart des langages proposent des librairies qui offrent ces types déjà programmés.

Pile

Pile

Une pile est une structure de données qui permet d'empiler des valeurs.

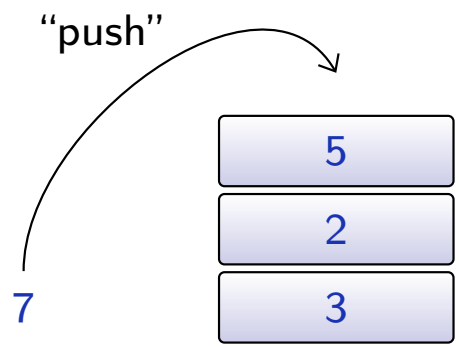
Les deux fonctions principales sont "Push" qui rajoute un élément au sommet et "Pop" qui l'enlève.



Pile

Une pile est une structure de données qui permet d'empiler des valeurs.

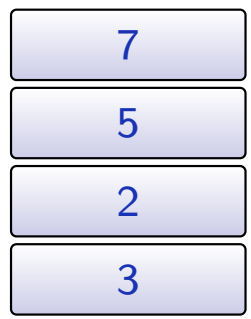
Les deux fonctions principales sont "Push" qui rajoute un élément au sommet et "Pop" qui l'enlève.



Pile

Une pile est une structure de données qui permet d'empiler des valeurs.

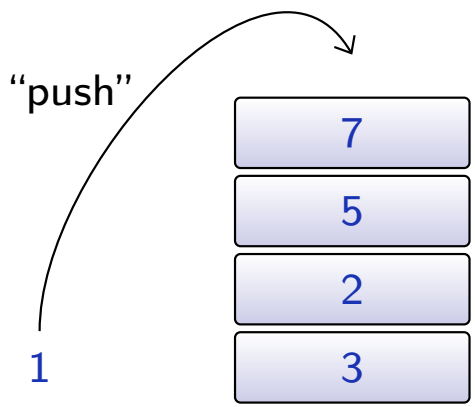
Les deux fonctions principales sont "Push" qui rajoute un élément au sommet et "Pop" qui l'enlève.



Pile

Une pile est une structure de données qui permet d'empiler des valeurs.

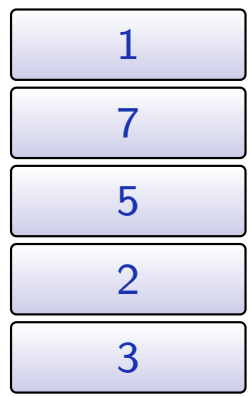
Les deux fonctions principales sont "Push" qui rajoute un élément au sommet et "Pop" qui l'enlève.



Pile

Une pile est une structure de données qui permet d'empiler des valeurs.

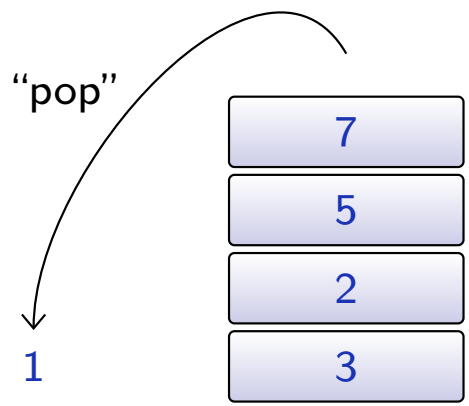
Les deux fonctions principales sont "Push" qui rajoute un élément au sommet et "Pop" qui l'enlève.



Pile

Une pile est une structure de données qui permet d'empiler des valeurs.

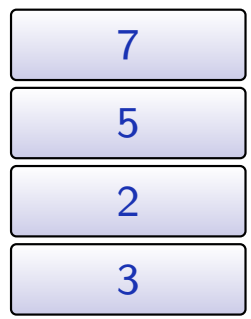
Les deux fonctions principales sont "Push" qui rajoute un élément au sommet et "Pop" qui l'enlève.



Pile

Une pile est une structure de données qui permet d'empiler des valeurs.

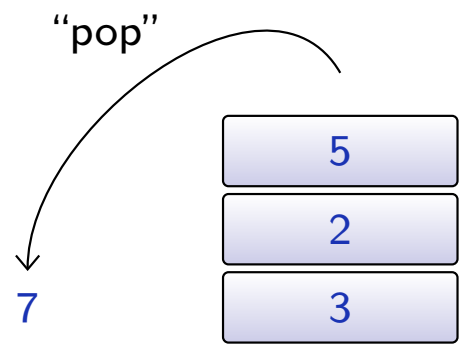
Les deux fonctions principales sont "Push" qui rajoute un élément au sommet et "Pop" qui l'enlève.



Pile

Une pile est une structure de données qui permet d'empiler des valeurs.

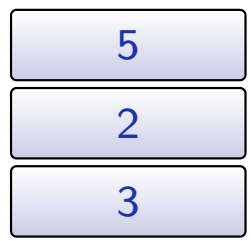
Les deux fonctions principales sont "Push" qui rajoute un élément au sommet et "Pop" qui l'enlève.



Pile

Une pile est une structure de données qui permet d'empiler des valeurs.

Les deux fonctions principales sont "Push" qui rajoute un élément au sommet et "Pop" qui l'enlève.



Pile (cont.)

Ce type de structure est très utile pour traiter des informations hiérarchisées (ouverture / fermeture de parenthèses, structures imbriquées), des historiques (“undo”, retour en arrière dans un browser web), etc.

Pile (cont.)

```
struct Stack<T> {
  var content: List<T>
}

func new<T>() -> Stack<T> {
  return Stack<T>(content: List.empty)
}
```

Pile (cont.)

```
struct Stack<T> {
  var content: List<T>
}

func new<T>() -> Stack<T> {
  return Stack<T>(content: List.empty)
}

func isEmpty<T>( _ s: Stack<T> ) -> Bool {
  switch s.content {
  case List.empty:
    return true
  case List.nonEmpty:
    return false
  }
}
```

Pile (cont.)

```
func push<T>( _ s: inout Stack<T>, _ t: T) {
  s.content = List.nonEmpty(t, s.content)
}

func pop<T>( _ s: inout Stack<T>) -> T? {
  switch s.content {
  case List.empty:
    return nil
  case List.nonEmpty(let value, let tail):
    s.content = tail
    return value
  }
}
```

Pile (cont.)

```
var stack: Stack<Int> = new()

push(&stack, 3)
push(&stack, 5)
push(&stack, 1)

for _ in 1...5 {
  print(pop(&stack) ?? "n.a.")
}
```

affiche

```
1
5
3
n.a.
n.a.
```

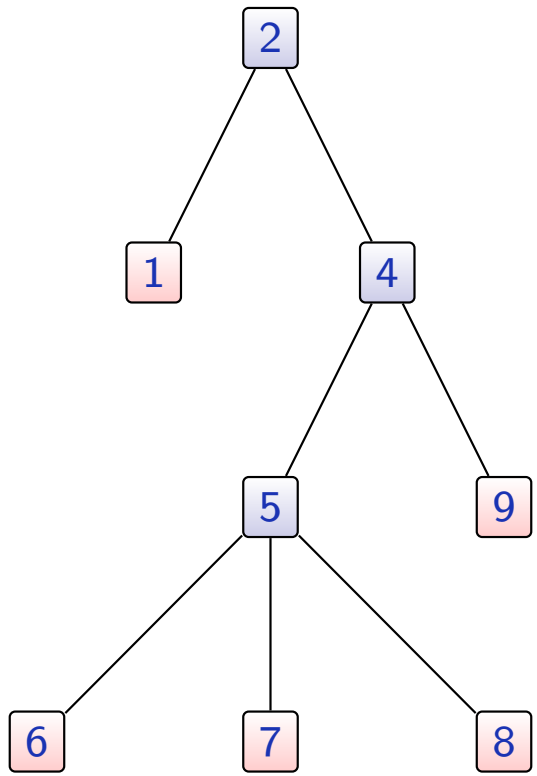
Arbres

Une autre famille de structures récursives très utiles sont les arbres.

On peut définir un arbre comme étant:

- une feuille, qui porte une valeur, ou
- un nœud (“interne”) qui porte une valeur et des références vers des [sous] arbres.

Arbres (cont.)



Exemple: Arbre généalogique

L'exemple le plus simple est l'arbre binaire, dans lequel un nœud n'a que deux sous arbres.

Nous pouvons considérer un arbre généalogique, dans lequel chaque nœud représente une personne, et peut faire référence aux deux parents.

Exemple: Arbre généalogique (cont.)

Une implémentation possible serait

```
struct Person {
  let firstName, familyName: String
  let birthYear: UInt
  var mother, father: Ancestor
}
```

Exemple: Arbre généalogique (cont.)

Une implémentation possible serait

```
struct Person {
  let firstName, familyName: String
  let birthYear: UInt
  var mother, father: Ancestor
}
```

où `Ancestor` peut gérer le cas d'un parent inconnu.

```
indirect enum Ancestor {
  case unknown
  case known(Person)
}
```

Exemple: Arbre généalogique (cont.)

```
import Foundation

func person( _ name: String, _ birthYear: UInt,
             mother: Ancestor, father: Ancestor) -> Person {
  let n = name.components(separatedBy: ",")
  return Person(
    firstName: n[1], familyName: n[0],
    birthYear: birthYear,
    mother: mother, father: father
  )
}

var bob_smith = person(
  "Smith,Bob", 1890,
  mother: .unknown, father: .unknown
)

print(bob_smith)
```

affiche

```
Person(firstName: "Bob", familyName: "Smith",
birthYear: 1890,
mother: family.Ancestor.unknown, father: family.Ancestor.unknown)
```

Exemple: Arbre généalogique (cont.)

```
var bob_smith = person(
  "Smith,Bob", 1890,
  mother: .unknown, father: .unknown
)

var claire_duchemin = person(
  "Duchemin,Claire", 1893,
  mother: .unknown, father: .unknown
)

var romuald_smith = person(
  "Smith,Romuald", 1914,
  mother: .known(claire_duchemin), father: .known(bob_smith)
)

print(romuald_smith)
```

```
Person(firstName: "Romuald", familyName: "Smith", birthYear: 1914,
mother: family.Ancestor.known(family.Person(firstName: "Claire",
familyName: "Duchemin", birthYear: 1893, mother: family.Ancestor.unknown,
father: family.Ancestor.unknown)), father: family.Ancestor.known(
family.Person(firstName: "Bob", familyName: "Smith", birthYear: 1890,
mother: family.Ancestor.unknown, father: family.Ancestor.unknown)))
```

Exemple: Arbre généalogique (cont.)

```
func printAncestor(_ a: Ancestor, _ indent: String) {
  let i = indent + "    "
  switch a {
  case .unknown:
    print(i + "<Inconnu>")
  case .known(let p):
    printTree(p, i)
  }
}

func printTree(_ p: Person, _ indent: String) {
  print(indent + p.familyName + ", " + p.firstName +
        " (" + String(p.birthYear) + ")")
  printAncestor(p.mother, indent)
  printAncestor(p.father, indent)
}

func printTree(_ p: Person) {
  printTree(p, "")
}
```

Exemple: Arbre généalogique (cont.)

```
printTree(romuald_smith)
```

affiche

```
Smith, Romuald (1914)
    Duchemin, Claire (1893)
        <Inconnu>
        <Inconnu>
    Smith, Bob (1890)
        <Inconnu>
        <Inconnu>
```

Exemple: Arbre généalogique (cont.)

Le nombre d'ancêtres connus d'une personne est calculable récursivement:

```
func nbKnownAncestors(_ a: Ancestor) -> UInt {
  switch a {
  case .unknown:
    return 0
  case .known(let p):
    return 1 + nbKnownAncestors(p)
  }
}

func nbKnownAncestors(_ p: Person) -> UInt {
  return nbKnownAncestors(p.mother) + nbKnownAncestors(p.father)
}
```

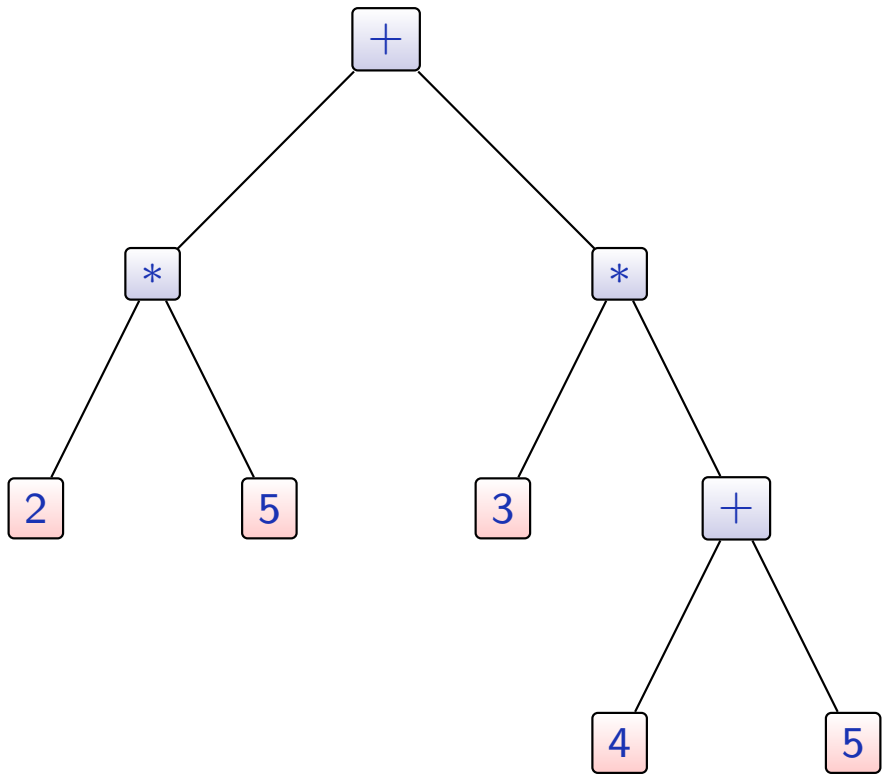
Exemple: Expressions arithmétiques

Un autre exemple classique où une structure récursive est très utile est la représentation d'une expression, par exemple arithmétique.

On définit une **expression** comme étant:

- Un nombre entier, ou
- la somme de deux **expressions**, ou
- le produit de deux **expressions**.

Par exemple $2 * 5 + 3 * (4 + 5)$.



Exemple: Expressions arithmétiques (cont.)

Cette définition se met directement sous la forme d'une énumération récursive en Swift:

```
indirect enum Expression {
  case number(Int)
  case addition(Expression, Expression)
  case multiplication(Expression, Expression)
}
```

Exemple: Expressions arithmétiques (cont.)

Cette définition se met directement sous la forme d'une énumération récursive en Swift:

```
indirect enum Expression {
  case number(Int)
  case addition(Expression, Expression)
  case multiplication(Expression, Expression)
}
```

Et l'évaluation d'une expression a la forme

```
func evaluate(_ expression: Expression) -> Int {
  switch expression {
  case Expression.number(let value):
    return value
  case Expression.addition(let left, let right):
    return evaluate(left) + evaluate(right)
  case Expression.multiplication(let left, let right):
    return evaluate(left) * evaluate(right)
  }
}
```

Exemple: Expressions arithmétiques (cont.)

```
let five = Expression.number(5)
let four = Expression.number(4)
let sum = Expression.addition(five, four)
let product = Expression.multiplication(sum, Expression.number(2))

print(evaluate(product))
```

Exemple: Expressions arithmétiques (cont.)

```
let five = Expression.number(5)
let four = Expression.number(4)
let sum = Expression.addition(five, four)
let product = Expression.multiplication(sum, Expression.number(2))

print(evaluate(product))
```

affiche 18.

Graphes

Les structures de données récursives se généralisent à des graphes quelconques. Cela permet de représenter des réseaux d'ordinateurs, de sites de production, de transports, de pages web, etc.

Les structures de données récursives se généralisent à des graphes quelconques. Cela permet de représenter des réseaux d'ordinateurs, de sites de production, de transports, de pages web, etc.

On pourrait par exemple imaginer un réseau social organisé avec un type

```
struct Email {
  let email: String
  var validated: Bool
}

struct User {
  let name: String
  let email: Email?
  let friends: [User]
}
```

Manipuler un graphe de ce type est complexe à cause des possibles cycles: X peut être dans les amis de Y et Y dans les amis de X.

Cela nécessite de travailler avec des références dans les structures de données, et d'être prudent quand on veut visiter ses composants pour ne pas avoir de boucles infinies.

FIN